Rubicon principles

Work in progress. This page is mirrored automatically from `rubicon/docs/principles.md` in the Causal Map repository and changes as the design does. The section written for whoever changes the code is not published here.

Rubric and Rubicon share a root, Latin *ruber*, red. A rubric was a heading written in red ochre in the margin of a law book, and the Rubicon was named for the red clay of its bed. Red means take me seriously. Crossing the Rubicon is what this app does. It's up to you to work out which river we are crossing.

What Rubicon is

Rubicon is an experimental wing of the Causal Map code base, with its own page and its own database. If it goes further it becomes a separate product. Its projects are Causal Map projects, so it can read the project's causal map, and each source's map, alongside the documents. But it has its own database and apart from sharing projects and their sources, there is not much connection between them.

It is a way to implement the proposed **open format** for qualitative text processing workflows in evaluation.

Rubicon is not finished, not priced and not yet publicly available, and the working name might change.

How it differs from Causal Map. Working in Causal Map is all about creating essentially one cognitive causal model of the ideas in the texts. From that model you can extract answers to many evaluation questions; but doing so is sadly not always straightforward. With Rubicon there is no overall model of all the texts. You answer each question in its own way, using a different workflow for each, or build one workflow using the results from other workflows.

How a piece of work goes

You pick a project, state a question, and argue out a workflow with Ruby before any analysis runs: which sources to read, how to mark them up, how to count what was marked, how to combine the results, how to reach a verdict. Then it runs, and each step's result sits under the step that made it. When one result answers the question, you accept it.

Fully specified means every decision settled before anybody sees the evidence. That is the hardcore end rather than a requirement.

1. State the question, and which part of it this workflow answers.
2. Name the sources, with a `sample` step.
3. Write the coding instruction.
4. Write the counting rule.
5. Write the rubric.
6. Validate it on synthetic texts.
7. Register it with the commissioner, dated, before the analysis runs.
8. Run it, and report what comes out.

Validating and registering are independent of each other: a registered workflow may be useless, and a validated one may never be registered.

Where people belong in it

The eight steps read as though one person writes a workflow and a machine runs it. That is the extreme case. Most evaluation worth doing has other people in it, and the places they belong are predictable enough to mark in advance:

- **Before step 1**, agreeing what the question actually is, which is usually a negotiation rather than a briefing.
- **At steps 3 to 5**, when somebody who knows the setting reads the coding instruction and the rubric and says which distinctions will not survive contact with the material.
- **After the first run of a coding step**, reading what came back, fixing the instruction, running it again. This is the loop the engine was built for: run one step, look, revise, continue.
- **At substantiation or member checking**, where the people described get to say whether the description is right.
- **At sense-making**, where a group reads the results together and decides what they mean, which is not the same as being told what the machine found.
- **At the verdict**, where a person may override it, and the override sits beside the machine's version rather than replacing it.

A workflow should say which of these it uses and where. Today it cannot: the engine can stop at a named step and continue from it, so the mechanism half exists, and there is nothing that marks a point as one where people are meant to intervene. That is a gap rather than a position.

What we claim, and what we do not

We claim no determinism and no full reproducibility. AI coding varies between runs, and machine agreement with human coders is well short of perfect. What survives is **auditability**: a reader who disagrees can find out exactly where.

Registration records ordering rather than commanding fixity, so revise as often as the work needs and never revise a version. The reasoning is under Worth is declared by a person.

Projects

You pick a project on the way in. Rubicon writes nothing back to it, so deleting Rubicon tomorrow would leave the Causal Map project exactly as it was.

What you land on is the project hub: every question anybody has asked of this project, what to do next on each, and a standing conversation with Ruby that is not attached to any single question and is shared by everyone on the project.

Sources

The project's documents, whatever was imported into Causal Map: interviews, reports, minutes, field notes. Rubicon reads them and never changes them, and it never treats one as something a workflow produced. A source is never a result.

Background documents

Some sources are the standard rather than the evidence: a theory of change, a funder's standards document, terms of reference. Those are flagged as background documents, so a workflow can read one when it needs to know what was promised, and nothing samples, codes or counts them as though somebody had said it in an interview.

The flag lives on the source itself in Causal Map, in a custom column, several spellings accepted, so one library serves every question you ask of that project. There is no button for it yet; you type it into the column.

A `sample` step drops them, so a workflow that samples first never codes or counts them, and a coding step with no sample in front of it still sees them. They are listed in the sample as left out, except where the same step goes on to narrow to passages. Ruby reads them while she drafts, up to twenty-four thousand characters, and is told to ask for the rest; no step reads them at run time. What she carries into the workflow is what she took from them: the columns, the levels and what they mean, the rubric rows.

Units

What gets counted, and what a figure counts across: a person, an organisation, a document. Every threshold needs one, because "more than half of stakeholders" has to know how many there are. Rubicon makes one unit per source and has no other way of making one, so a project whose units are really people, with several interviews to a source, cannot yet say so.

Chunks and segments

A long source is cut into equal chunks of at most sixteen thousand characters, one model call each, unless an earlier step already narrowed it to passages, when each passage is a call.

The whole source is pre-swept into numbered segments and each chunk carries the markers for the ones it holds, so a number means the same thing in every chunk. That numbering is the handle for making a model account for the whole of what it was given rather than stopping at the first good thing.

Segment accounting. Ask a model to find what matters in a long transcript and it finds a few good things and stops. Requiring it to account for every segment, with a refusal that has to quote the segment and say in a few words why it will not do, makes declaring a segment empty the expensive option. `segment_quota` and `min_per_segment` are the dial, and coverage is reported on the asset.

Passages and the index

A step can ask to read the passages about something rather than whole documents. What it searches is Rubicon's own index: each source cut into pieces of about twelve hundred characters, each piece knowing the character range it came from, turned into a vector and stored beside the runs rather than in the project. The pieces are small on purpose, because a passage is judged on whether it is about the query and a long one dilutes whatever made it match. The character range is what makes the rest work: a quotation found inside a passage still resolves against the source, and what the next step reads is a window of the surrounding text rather than the retrieved piece alone.

Sources are embedded the first time a step needs them, and only the sources that step will search, so narrowing can be asked for on a project nobody has prepared. One whose text has not changed since it was embedded is left alone and one whose text has changed is embedded again, which the content hash decides. The embedding is charged to us and does not yet appear in any cost figure.

A check buys none of it. It is free by contract, Ruby runs one on every draft, and it undoes its own writes when it ends, so embedding there would pay for vectors nobody keeps, once per draft, on a corpus of any size. It reports the step as needing a run instead, saying what running it would embed. The index is bought once and every later run searches it without paying again.

Boilerplate. A passage appearing in nearly the same words, in nearly the same place, in three or more documents. Found by repetition rather than by recognition, so the test holds for a form nobody here has seen. Wording counts for 0.7 and position for 0.3, scaled by length, and anything under ninety characters is never considered. It is marked rather than removed, and left out of a meaning search unless a workflow asks for it: a form header naming the programme, the roles and the subject is faintly similar to every query about the corpus and outranks the passages that answer one. Asking for it is worth doing, since a question everyone was asked and nobody answered is a finding.

Questions

A question is what you want answered, and everything else hangs off one. A project usually has many and they are the unit of work: you ask one, argue out a workflow for it, run that, and accept one result as the answer. Several workflows can attempt the same question, and asking the same thing twice does not make a second question, because every run files itself under the question's own text.

A question is answered exactly when a result stands as its answer, which is read off the answers rather than set by anybody. So nothing can mark a question answered while nothing stands as its answer, and a report can list the questions that have none.

A report gathers the accepted answers of several questions into one document, a section each, and names the questions that have none.

Well-defined

Two careful readers would apply the coding instruction much the same way, and every row leads back to a quote a third person could check. Being well defined is what Ruby refuses a question for, and nothing else is.

- A property of the question and the material together, usually achieved rather than discovered. Most questions arrive ill-defined, and making them well defined is Ruby's job.
- Not the same as objective, factual, measurable or important. A well-defined question can be trivial, and an excellent question can be ill-defined.
- A matter of degree rather than a line. What carries it is the question, which is why checking the answer afterwards is a different thing: an answer can be traceable, checked and sound, and still answer a question nobody asked, because that was the one the reader could see how to answer.

What gets refused, and what does not, is under What it refuses, and what stays soft.

Accepting an answer

Naming one result as the question's answer. It points at a result rather than at a run, so a table can be accepted as readily as a judgement. One answer stands at a time, which the database holds as a unique index rather than a careful function, and a question is answered exactly when one does.

Archiving

Hiding runs, or a whole question, when you are finished with them. A workflow shows as archived once every run of it is. It never deletes.

Ruby

The assistant. She reads the corpus, says what she takes a question to mean, argues with the parts that will not work, and proposes a workflow. She never decides what counts as good.

- **Ruby is the way into the app** rather than a tool beside it.
- **She learns what the app accepts by being refused.** Refusals are written to teach, and are not written out for her separately, because a second copy would be free to drift from the code enforcing it.
- **She is a methods coach rather than an order-taker.** A question can be perfectly well defined and a poor use of the material: it rests on two speakers out of seventy, or it counts how many said something where the interesting thing is what they said. She says what she would ask instead, then writes the workflow for what you actually asked. The evaluation is yours rather than hers.
- **What gets agreed is the workflow** rather than the conversation.

She can look at what has run. Before anything has run she knows the corpus by its names and columns and not a word of what the documents say. Once runs exist she can ask to see what they produced by naming it in `look`: what a run did and made, one result in full, a coding's rows or a narrowing's passages a page at a time, a step's model calls and what came back, or the text around a quotation. Every page says which rows it holds and of how many, so a page is never read as the whole. She can look at any run on the project and at nothing outside it. Rounds of looking are bounded apart from the turns a draft needs; the turn's cost bound holds over both. What she looked at is recorded on whatever she writes next, because a standard written after seeing the results can be made to fit them: where she proposes or changes a rubric after looking, she says so, and where its thresholds sit stays the evaluator's call.

Conversations

An exchange with Ruby about a question. A question has many and a workflow has many, neither owning the other. It is a run like any other and carries a status, but what it produces is a workflow or nothing, so that status says whether the conversation ran and never whether the workflow is any good.

Rewriting a turn. Edit anything you said and the chat carries on from there, dropping the turns below. The result is a fork: a new conversation that reads the earlier turns from the old one and holds none of the later ones. The original is unchanged, and nothing is copied, since the fork stores only the turn it was cut at.

A conversation is public where the project is. It sits under the same predicate as any other run, so a collaborator sees it and so does anybody at all where the project is public. That is the point rather than an oversight, and the reasoning is under The ledger, and who may write.

Drafts

A workflow Ruby has written and you have not yet run. Read it, argue with it, change it. Checking that it will run costs nothing, and nothing is spent until you press Run.

Workflows

One approach to a question: a set of steps, the graph joining them, and a name. One question may have several, and three workflows on one question are three different ways of answering it rather than three attempts at one.

- **You never number the steps.** Each step says what it needs and what it makes, and the order falls out of that. Two lines of work can run side by side and come together later.
- **The name is the one part somebody chooses.** It says which way of answering the question this is, Ruby is asked to say how each one differs from the others, and a workflow with no name is refused.
- **A workflow's state** is one of: not run, running, waiting, failed, candidate, accepted, archived. What the project hub shows against a question, what to do next, is read from its workflows rather than stored: ask it, read what Ruby said, run the draft, watch a run, read what came out, or it is answered.

Versions

One complete statement of what to do, under a workflow's name. Everything that decides what the answer should be is part of it: the steps, the documents it will read, and the code that will read them. Change any of those and it is a different version, because it is a different thing to do. The shared name is what says two versions are refinements of one approach.

- **A version is named by what it is.** Its name is six characters derived from the specification itself, `f35oae` rather than `v3`. The same recipe always gets the same name. Two people cannot claim one name for different work. Nothing has to be remembered or bumped. Siblings do not compete for a number.
- **Versions form a tree rather than a line.** Two refinements of one approach are siblings under one name, and neither supersedes the other. So `foo f35oae` and `foo 7d33ae` are two attempts at one idea rather than unrelated ideas, and a run that fails and is fixed produces a new version by being different.
- **The test that separates a version from a run.** Did somebody change what should be done? A new version. Did only the getting-there change, a dropped connection or a killed process? Another run of the same version.
- **Two of the three move without anybody deciding anything.** Documents arrive, are deleted or are edited, and we fix the engine. Both change what the answer should be, so the page says which moved rather than leaving somebody to wonder why the verdict did.

Registration

Fixing the method before the evidence is read, with a version, a date and a named person. What is registered is the whole workflow: the sample, the coding instruction, the counting rule and the rubric alike, since most of the judgement lives in the parts that are not the rubric.

Pressing Run is the act of registering it, and vouching for the method. The record carries who ran it and when rather than a justification. If you don't get it, don't do it.

A registered part is versioned, never rewritten in place, and recorded on each run as the version that was in force. Why a declaration of worth needs that trail when a count does not is under Worth is declared by a person.

The workflow file

YAML: a header, then a list of steps. `rubicon/workflows/` holds worked examples, and `example-who-was-interviewed.yaml` is the smallest, spending nothing.

```
id: x-versus-y            # the name: one per way of
answering the question
version: 1                # a readable ordinal; the
version's real name is derived
project: bpea-evaluation-copy
question: >
  Did the programme create access academics could not
otherwise have reached?

steps:

  - id: frame
    type: sample
    spec: {method: random, n: 2, seed: 7, where:
{s_Type: ["Academic interview"]}}
    outputs:

      - {name: sources, role: source_set, type:
source_set}

  - id: find_evidence
    type: code
    inputs: [sources]                  # names, or
roles, of earlier outputs
    spec:
      model: gemini-3.8-flash
      chunk_size: 3000
      instruction: >
        ...what to look for...
      columns:

        - name: favours
          type: category
          means: which reading this passage supports
          values:

            - {name: new_access, means: "...what earns
this value..."}
    outputs:

      - {name: evidence, role: evidence, type: quotes}
```

The format itself is published, and is not restated here. `webapp/rubicon/spec/0.1/` holds the schema, the conformance suite and the requirements in RFC 2119 words, served at the URL the schema's own `$id` claims. Which keys exist, their types, their enumerations and what may not appear beside what are all stated there and checked against a draft before it runs. `shape.reference()` renders them, which is what Ruby is given and what to read when writing one. A table of keys in prose would be a second copy, free to drift from the code, and the drift would be invisible: a key that stopped being read looks exactly like one that still is.

What this page carries is what a schema cannot: why a refusal exists, what a step does with what it is given, and everything about behaviour rather than structure.

`id` and `version`. `id` names the approach. `version` is an ordinal you may write for a reader; it identifies nothing, so two versions may carry the same number without colliding.

`inputs`. The names or roles a step reads. Order comes from these and never from position in the file, so two branches can run side by side and meet later. `rubicon plan <file>` prints the order the engine derived.

`means`. The sentence saying what a column, or one value of it, stands for. Required on every column and on every level of a category or an ordinal. A column whose levels are bare names is refused, because a model choosing between undefined words is consistent only by luck.

`execution` **and** `tools`**.** `execution: agentic` lets a step take several turns and use the tools it lists, bounded by `max_turns` and `max_cost_usd`. Without it a step is one call.

`from_run`**.** An input may read an asset another run made, written `{from_run: <run id>, name: <asset name>}` in place of a plain name. It is how a question about results this project already holds is answered without making them again: comparing two earlier answers, rolling several up, judging against a standard applied before. Ruby is given the run ids on the project's earlier questions and this syntax beside them.

- **A DIFFERENT workflow's results, never this one's own.** Reaching back into an earlier run of the same workflow is refused. The continuation inherits every finished step without naming any id, so a new version is written out in full, the finished steps are skipped rather than paid for again, and the reader sees the whole workflow. A version written as the new step alone, pointing at the old run, answers correctly and is not a version of that workflow: opening it shows one step, and what the answer rests on is elsewhere.
- Nothing checks that the other run used the same corpus or the same settings, and pulling one in widens the sources an agentic step may reach, so the comparability has to be asserted in the workflow's own words.
- It costs self-containment, which is the reason to reach for it rarely. A workflow naming a run id only works where that run lives: it cannot be copied to another project, shared as a worked example, or trusted once the run is archived.

The step kinds

One instruction inside a workflow. It says which results it reads and what it produces, and it is the smallest thing that can succeed or fail on its own. Eight implementations under ten names. A new one is added by writing it and importing it in `steps/__init__.py`; there is no separate list to keep in step.

Type	Reads	Produces
`sample`	the project's sources	a `source_set`, or `passages` where it narrows within the documents, and a sample every later asset carries
`code`	a source set, or passages	`quotes`, one row per marked passage
`compute`	quotes, or a sample alone	a `table` of named figures
`align`	two or more codings of the same text	`quotes`, one row per aligned passage, for a later `compute` to count
`judge`	tables and quotes	a `judgement`
`note`	anything	a `note`: prose with its quotes attached
`theory`	the counting step's table, the judging step's verdict, or a coded table	a `theory_of_change`: the links, with the figures and the verdicts hung on them
`synthesise`	a rubric and background documents	a `source_set` of generated documents, each carrying its expected verdict

`lane` **sits in any single-pass step's spec**, `sync` by default and `batch` for the cheap slow route, with a run-level default behind it. It is refused on an agentic step by name, since a step that decides its second prompt from the answer to its first has nothing to send ahead. `code` is where it earns its keep, being one call per chunk per document.

sample

Say what may be read next: over documents, over passages, or both. `select` and `select_passages` are aliases the engine still accepts and the format does not, and they are to be removed; `kinds()` in `steps/base.py` already collapses them so an author is not shown three names for one thing.

What it produces is a **sample**: which sources were read, out of which set, chosen how, and a fingerprint of what each one said. A draw that leaves anything to chance states the seed it drew on, so the same version run twice reads the same documents. The fingerprint is what makes an edited document visible: the ids alone are the same before and after somebody rewrites a transcript.

Frame. The set a sample was drawn from, with a `frame_description` a reader can check.

Method. `all`, `random`, `stratified` or `purposive`. A stratified draw is even across the groups one source column makes, or across the combinations several of them make, since an even draw across province and sex is a different sample from an even draw across either alone. It is refused where a column is one the sources do not carry, where most of them leave it blank, and where the columns make more groups than the draw reads documents, all three for the same reason: a draw that cannot be even should not come back wearing the word. A step reads the whole frame unless it is told to narrow, so a sample is something you ask for or Ruby proposes, never something that happens on its own. She proposes one when the reading would be expensive: a draft costing more than forty model calls is refused back to her until she justifies it in writing, and cutting it to a sample is the usual answer. Nothing checks again when Run is pressed. A `random` or `stratified` draw with no seed is refused, because a draw nobody can repeat is not evidence.

Exclusion. A rule over the source columns with a reason attached, kept beside the count of what it removed. "Forty-two of fifty-three sources, after excluding eleven because the pilot is not part of this evaluation" is built into the sample's own description. Refused without a reason, and refused when it matches nothing.

Narrowing by meaning. `relevant_to` hands on the passages about a question rather than whole documents, out of the index. `threshold` sets how close a passage must be, `per_source` and `top_k` how much comes back, and `before` and `after` widen each hit into the surrounding text, since the sentence naming a cause is often beside the sentence naming the outcome rather than inside it. Three things follow, and a workflow using it has to say them. What is passed on is the passages, so every later step sees only those, which is the subject of Say what you did not read. `per_source` is a cap rather than a filter, three by default above twelve sources, so a document with ten passages on the subject contributes three and no count after it can compare how much different documents said. And the query decides what can be found, so narrow on the subject the question is about and never on the answer it is testing: a question asking whether the organisation contributed to family wellbeing compared with other influences retrieves on wellbeing and codes whatever causes it finds, where retrieving on the organisation would return the passages that mention it and settle the comparison before any reading began.

Ask it the way a respondent would say it, and ask it more than one way. A meaning search compares one text with another, so the query is itself a text rather than a set of terms. Written as a sentence somebody might have spoken, it lands among the passages that speak that way. Written as keywords strung together, "food security diet harvest hungry season food production", it lands at the average of six subjects. No passage occupies that place, so what comes back is the material vaguely about all six instead of the material answering any one of them. The keyword habit is worth naming because it looks like thoroughness. More terms feel like wider coverage; in a meaning search they narrow it to a blur.

The way to get the coverage the keywords were reaching for is to ask several questions rather than one wide one. `relevant_to` therefore takes a list, each phrasing searched on its own and the results combined by rank rather than by score. Combining by rank matters: a similarity is not comparable between two queries, so adding the numbers up lets whichever phrasing happens to score high everywhere decide the whole result, while ranking asks each phrasing only to order its own findings and then rewards the passages that several of them reached. What surfaces is what different wordings agreed on, which is close to what an evaluator means by a passage being relevant. The record carries every phrasing and how many passages each one found, because each phrasing decided part of what could be found and a reader disagreeing with the selection needs the whole of what was asked.

A question and a passage are not the same kind of text. The embedding model is told which of the two it is looking at: `RETRIEVAL_DOCUMENT` for the corpus, `RETRIEVAL_QUERY` for the thing asked of it. The two sides land differently. The default is the query side, so an index built without saying anything was built as though every passage in the corpus were a question somebody had asked. Because the recipe is part of what a vector means, the index records the model and the task type together. A source embedded under an older recipe then counts as not indexed: it is embedded again rather than compared with its neighbours. Vectors made two different ways do not fail when they are compared. They rank, plausibly, with nothing downstream able to tell.

Nothing marks a sample as a trial rather than as the analysis. A run records that it was sampled and never why, so two sources read to see whether a category holds and a considered draw of twenty look the same in the ledger. Piloting today means stopping the run at a named step from a terminal and reading what came back; there is no button for it, and the case for one is in `rubicon/xkTODO the pilot discipline needs a button xpCLAUDE.md`.

code

Mark up the text. A pass that finds passages and records whatever you asked about each one, in columns declared in advance. Its rows are quotes.

- Causal Map has one canonical coding per project, because it builds one model of the sources' causal cognitions. Rubicon has no such object. A coding answers the question its own instruction asks, so a project holds as many codings as it has questions.
- **A coding may cover less than the whole text.** A chunk whose answer comes back unreadable is recorded as unread and the pass carries on, so one reply in two hundred no longer takes the other 213 with it. The count is on the asset beside the segments a model declared empty, so a figure built on this coding can state 23 of 24 chunks rather than implying all of them. Past a share the workflow declares, the step refuses instead: a model that has stopped answering must not produce a table built on a tenth of the text while looking complete.
- **All rows refused.** A coding whose every row failed the declared columns is refused, rather than recorded as having found nothing.
- `chunk_size` and `segment_quota` decide how much text the model weighs at once and how much it has to find per segment. Halving the chunk finds more of the same kind of thing, which is why two codings are only comparable when both were given the same budget.

Composites. One object assembled from passages that share a declared key, each part keeping the quote it came from. `composes: {parts: [context, reasoning, outcome], key: <column>}`, and what it makes rides on the coding's own asset. A realist configuration is the case it was written for, and the same arrangement turns up wherever an object is built from evidence scattered about, such as an outcome statement with its actor and its substantiation. It is not a result in its own right, so it cannot yet be accepted as the answer to a question or drawn.

The key has to be a declared column, and that is the whole of the discipline. Parts assembled because a model said they belong together is a claim with no trail, and a reader could not tell a configuration somebody found from one somebody composed. A composite whose parts do not all appear is kept and says which are missing, because a configuration with no outcome is a finding about the material rather than a broken row.

compute

Work out a figure. The counting language is under Working with numbers, below.

No reading required. "How many households in each province, by age band" is two columns the sources already carry, and a `compute` given a sample rather than a coding answers it from those, calling no model and opening no document. Look for that before proposing anything that reads text.

Several tables from one coding. `compute` takes the same rows and gives a share per case, a count per actor type, a breakdown by year, and the same figure again restricted to speakers with no stake in the programme. They are separate figures with separate bases rather than one number carrying caveats, and a share over a frame the draw cannot speak for is refused rather than printed.

align

Group two or more codings of the same text, so both the agreement and the silence can be counted. A group records which codings covered the passage and which never marked it.

Testing the instrument. Group two runs of one instruction with `align` and the flip rate falls out of it. The second run needs `reuse: false`, or it inherits the first one's answers and the flip rate comes back at nought for a mechanical reason, which `align` reports as a concern in its own words.

It checks that two codings had the same opportunity: same sources, same settings, comparable segment coverage, and no shared reused answers. It reports the disparity on the comparison and refuses nothing.

judge

Apply a rubric and reach a judgement.

note

Write it up: prose with the quotes it rests on attached and enforced. A note that cites nothing is a negative finding, and its evidence is the search rather than a quotation, per Every claim points at a quote.

theory

Hang the counting and the judging on the theory of change a workflow declared, and file the annotated theory as a result in its own right. It reads the links from the `for_each` that tested them, with `links_from`, so the shape is stated once, and follows that back to the asset the fan-out read where the list came from one.

Where a workflow declares no links at all, it reads them out of a coded table instead, taking each distinct pair of ends as a link and counting the passages behind it: that is a theory of change read out of the material rather than checked against it, and it is what Causal Map has always made. This is the thing a contribution analysis is for, and it is not the margin map.

synthesise

Make text to validate a method on. Showing that a workflow works at all, by running it over synthetic texts whose judgements are known in advance.

Its refusals are the method: it will not generate from the rubric alone, it will not take a set with no decoys unless you say why, it demands an expected verdict on every document, it will not default the generating model, and it namespaces what it makes as `syn:` so it cannot be mistaken for a real source.

The last step is missing, the one that compares the verdicts awarded against the verdicts expected and reports the separation as a figure. Design in `rubicon/validate a workflow on synthetic contrasting texts xkTODO xpSTEVE.md`.

Fanning a step out over a list

A theory of change with twenty links is twenty questions to put to the material. One pass with twenty columns gets a model that finds a few and stops, and each proposition needs its own count.

`for_each`. Written beside the step's `spec`. Four forms:

```
for_each: [a, b, c]                              # the list,
written down
for_each: {from: hypotheses, field: text}        # a field
of an earlier step's asset
for_each: {values_of: s_Region}                  # every
value that column takes, over every source
for_each: {values_of: actor, of: harvest}        # every
value a CODED column took, which is the
                                                 # grounded
way to fan out over something the
                                                 # material
produced rather than something typed
```

Item. One entry in that list. Substituted into the spec as `{item}`, or field by field as `{item.text}` where it is an object, which is what lets a theory of change declare each link as `{id, from, to, text}` and be drawn afterwards as the graph it came from.

Item run. One copy of the step, running on one item. Its step id carries the item, lower-cased and slugged, in brackets: `test_link[gap-needs-agency]`, and its outputs take the same suffix: `link_evidence[train_h]`. Item runs are built from the same inputs and never receive each other's outputs, which is what makes them safe to batch, resume one at a time and gather. Where item run two needs item run one's answer, that is two steps.

An item run says which item it is about in its spec or in its inputs. `{item}` in the spec is the usual way, most often inside a `where`. Where a step has nowhere in its spec to put it, it names the asset that prong should read instead: `figures[{item}]`, or `figures[{item.slug}]` where the asset came from an earlier fan-out. That is what lets a `judge` fan out at all, its spec being a rubric, a model and two ceilings with no filter anywhere in it, and it is how one judge per partner organisation reads that partner's own figures.

Gathering. A later step naming either the role or the `for_each`'s declared name, and receiving all the item runs at once. The fan and the condensing stay apart, because welding them would put one idea of the answer into the app.

Where a `compute` does the gathering, the item each item run ran over arrives as an ordinary column: `over: {group: item}` gives a row per proposition, `over: units` gives one figure across the documents with every item run's rows pooled. The full set of items is known from the assets, so a proposition the material never mentioned reads as a nought rather than vanishing, though an item run that failed outright vanishes with it. Item runs coded to different columns are refused rather than pooled. `rerun` names one step and cannot name a `for_each`, because the ids in the queue carry their brackets.

Four refusals. A `for_each` written inside `spec`, because nothing reads it there and the step would run once while the workflow read as a `for_each`. A `for_each` over an empty list, which never ran. A list longer than `max_items`, fifty by default. And a `for_each` over more than one item that mentions `{item}` in neither its spec nor its inputs, because its item runs would be copies: a crosstab drafted that way runs twice over the whole corpus and files two identical tables under two province names, every step reporting success.

Working with numbers

QCA's consistency, an outcome harvest's share of actors and an evidence rubric's agreement figure are the same arithmetic wearing three names, so none of them has a function of its own. Write one called `coverage` and a method's meaning has moved into the app. Instead there is one small language, used by every `compute` step.

```
{reduce: sum, of: {min: [x, y]}, over: units, where:
{stance: supports}}
```

For each unit that gets past the filter, take the smaller of x and y, then add those up.

Base. The items a figure considered, built before the filter rather than from the rows that survived it. A source read and found empty stays in the base, which is the difference between nobody saying it and nobody being asked.

`over`. What you count across. Five values and no others.

- `quotes`, or `rows`: coded rows.
- `units`: whatever the coding read, or fewer where a `sample` narrowed the corpus first. Units that produced nothing are in the base.
- `sample_frame`: the frame the sample was drawn from. Refused where only part of the frame was read and the draw was not random or stratified, and refused where no sample is among the step's inputs.
- `{group: <column>}`: a figure per value, seeded from the values the column declares, so a group that got nothing reads nought under `count` and nothing at all under `share`, `mean` or `median`. A `text` or `number` column declares no values, so nothing is seeded.
- `{group: [<column>, <column>]}`: a cross-tab, outer column first, every cell present wherever both columns declare their values. One figure and one table, never a `for_each` producing a table per value.

With a coding among the step's inputs a group counts coded rows, and without one it counts documents. The same expression means two things and only the figure's name says which, so name it for what it counts.

`where`. Which of them count at all. A plain value means equality, a list means any of them, `{contains: "..."}` matches text case-insensitively, `{not: value}` excludes, `{at_least: level}` compares on a declared ordinal's own order.

A `where` also sits on the step's spec, beside `figures` rather than inside one, and says what the whole table is about: this trade, this province, this year. It is folded into every figure the step declares, reaching both halves of a ratio, so a share narrowed on top and left wide below cannot be written. A column the step and one of its figures both filter is refused rather than merged, because letting one silently win produces a figure that reads as narrowed by the other, and because two different clauses on one column would match nothing and report a nought.

`of`. What to take from each item. It nests, as `min` does above.

`reduce`. One of `sum`, `count`, `mean`, `median`, `max`, `min`, `share`, `any`, `all`.

`ratio`. A fifth top-level form, holding a numerator and a denominator, each written with the four parts above.

What it refuses, all of it aimed at a number that would look finished and be wrong: an unknown operator, by name; a column the data does not have, wherever the filter sits on the expression itself rather than nested inside `of`; a column name resolving to nothing; a mean of a category or a text column; `at_least` on a column nobody declared as an ordinal; and a ratio whose halves counted different sets, unless you give `bases_differ_because`. Nothing is parsed from a string, so nothing can be mis-parsed. A zero denominator returns no value with the reason attached, since an absent denominator is a fact about the data rather than a fault in the expression.

Rubrics

A scale whose levels are ranked by worth: each row a verdict, what it stands for, and what earns it. It is how you get from "five out of seven" to "adequate".

The rubric belongs to the workflow, written inside a `judge` step's spec, so it versions with the steps and a registered workflow determines the standard it will apply. A bare name instead of a rubric names one registered against the project, which the engine resolves to the latest version of that name: an old workflow's standard can then change under it, which is why the format deprecates the form.

- **The verdicts are whatever you say.** Nothing in the app knows which set you will use, or assumes red, amber and green, or three levels.
- **Criteria.** One judgement may rest on several: salience, strength, frequency, coverage, whatever the question needs. Two criteria in one rubric need not share a set of levels: one can be red, amber and green while the next runs 1 to 5. A band is a contiguous run of levels, such as "adequate or better", and it only means anything where the levels are ranked.
- **Where one description fuses several criteria** and you would rather keep them apart, the column splits into one per criterion. Rubicon takes either.
- **Some workflows never use one.** A step may produce a yes, a shortlist, or a short story written to make a typical account palpable, without reaching an evaluative judgement.

Its keys are in the schema. Verdicts, criteria, standards, rules, `evidence_from` and `overall` are all stated there and rendered by `shape.reference()`, so they are not restated here. What follows is what the schema cannot say.

- **A criterion counts as arithmetic as soon as any one of its standards carries a rule.** Standards are tried in the order written and the first whose rule passes wins, so write them strongest first. A standard without a rule is passed over, and an unmatched criterion falls through to the last standard, whether or not that one has a rule.
- **Combining reads the order the verdicts were declared in, best first.** That is only safe where somebody declared them on purpose, so a rubric whose verdicts are inferred from its standards is refused, and so is a criterion whose verdict falls outside the declared set. Ranking a verdict nobody declared would turn an unresolved criterion into a confident answer.
- **A rubric that keeps its criteria apart must say how they combine**, because whether one red outweighs two greens belongs to whoever wrote the standard. One that says nothing produces a judgement that reports each criterion and states that nothing combined them, rather than inventing a verdict.
- `evidence_from: run` answers a criterion from the run's own record, the workflow as executed and what each input asset reports about its own making, rather than from the material. It is how a criterion about the method is judged without quotations, the other way being a rule over the figures the counting already produced.
- **A `judge` step is refused if the rubric it names does not exist.** On the page, a rubric Ruby drafted is registered by the act of pressing Run. Approval is that act rather than a separate ceremony.

Two ways to say how criteria combine. Standards test the set: every criterion green, no criterion red. Or `combine`, which reads the levels as a ranking. `combine: worst` is the chain, as strong as its weakest link, so a theory of change running A to B to C fails at whichever link the evidence does not support, however good the others are. Standards cannot express that, and an average, which is what a report reaches for when nobody wrote the rule down, hides it. `combine: best` is the branch, where two routes reach the same outcome and the better route carries the claim. Nesting them is not supported, so a chain of branches is written as branch summaries combined with `worst`.

Tabulate the judgements before rolling them up. Twelve tests judged one at a time against one rubric give twelve verdicts in a table, each carrying its own quotes. The roll-up is a second `judge` against a second rubric, which states in advance how the parts combine.

Challenging the question. A criterion may refuse the question instead of awarding a verdict, per What it refuses, and what stays soft. One that does sinks the overall verdict rather than joining the vote.

Splitting the corpus

A `sample` step decides what may be read, and a run records which sources it actually read. Nothing yet refuses to score a test on the material that produced it; the record makes it visible, and the assertion is a gap.

Runs

A version with its results filled in, where there are yet any. One go at it, which may not finish, so it also answers how far it got, what it cost and whether it crashed. A version run three times is one workflow and three of these.

The results are part of it rather than beside it. A run handed to somebody is the workflow as executed with each step's outputs written in under it. Nothing points out of the bundle: a figure carries the set it counted over and how big that set was, a quote carries its source and the character range it sits at, and every result names the results it was made from. So lineage is not a second document to keep in step; it is what the run already says about itself. That is what makes a run the thing you hand somebody: a reader needs no second document and no account from us.

It records order, results, model, duration, cost, who started it, the version it ran and the engine that ran it, and every model call with the prompt as sent and the answer as returned.

Running twice is repair or a reliability check, so runs are a version's history rather than things to compare. What tells two of them apart is when each ran, how it ended and what it cost. Never the instructions and never the sample, which belong to the revision: changing either makes the next version rather than another run.

Running a step at a time

A step at a time is the intention: run the coding, read it, fix the instruction, run it again, and only then go on to the counting.

- `until` stops at a named step. `continue_from` picks up what an earlier run made, recording the inherited steps as skipped. `rerun` does named steps again. On the command line, `--until <step>` when starting and `resume <run-id> --rerun <step>` afterwards.
- `rerun` names steps, and cannot name a `for_each`, whose steps are called `test_link[...]`, one per item run. Name the item run.
- A run that was told to stop records `cancelled` rather than failed. The request is written into the run's row by `rubicon.stop_run` and read before every model call, so a run stops within one call rather than at the end of a step. It keeps what it finished and can be continued.

What stops a run that nobody stopped

Three things.

- **A ceiling on model calls**, derived from what the run was estimated at rather than set by hand: twice the estimate plus ten, so a document the chunker misjudged does not halt a run estimated at two calls, while a runaway is caught well inside the damage. The estimate itself is never used as the limit.
- **The allowance of the account that started it**, read from `public.get_user_ai_credits_status`, which is the one place that decides. A credit check that fails to answer lets the run carry on, because stopping somebody for want of an allowance nobody could read is the worse error.
- **Two runs at once per account**, since a run holds its place from start to finish and a coding step sends several prompts inside that, so two is already a dozen calls in flight against a shared Vertex quota that has gone capacity-dead before with no incident posted.

The first two end a run the way a stop request does, keeping what finished and leaving it to be continued; the third refuses the start instead, and says which runs are already working so somebody can wait or stop one. The allowance is the one a user actually meets, and it costs them what has not run yet rather than what has.

Budgets bound an agentic step, `max_turns` and `max_cost_usd`, read on the same path as the stop request.

Reuse and inheriting

A run is built to repeat, so most of what could vary is held still. A draw states its seed, and a step whose call has been made before inherits the answer, so a second run over an unchanged corpus can finish having spent nothing. What is left free to move is the material, because sampling reads the project as it stands and a source added since widens the frame, and the models themselves wherever the text has changed or a step has turned reuse off.

Reuse answers a step from an earlier identical call rather than paying again. On unless a step sets `reuse: false`, keyed on the request as sent with the source text inside it and scoped to project and model, so an edited document cannot inherit an old reading. Turn it off where repeating the work is the work: a second coding run measuring how much the answer moves would otherwise show none. Every reused answer is recorded as such, with the earlier answer it came from, so a run that reused everything and a run that called nothing never look alike.

Inheriting a step is separate from call reuse, and coarser: what is skipped is the whole step rather than the model calls inside it. Two routes. A run started with `continue_from` reads what an earlier run made, and where nobody names one, the last run of the same workflow on the same project is assumed; what carries over is decided step by step, the same step by spec hash having read assets with the same contents. Failing that, the project is asked whether this exact step has ever finished, under any workflow, over inputs with the same contents, and what it made is copied in with every row saying which asset it came from. So a workflow with a new id still takes an identical step it happens to share with an older one, and a new version redoes the steps it changed and inherits the rest. `rerun: [step]` and `reuse: false` each insist on the work being done anyway. The gap this leaves is a step that differs in any particular: a reworded instruction or a different chunk size is a different step, correctly, and does all its reading again.

A step that reads the project is never inherited. A `sample` step looks at the world rather than at an earlier result, so its spec and its inputs are identical whether or not somebody edited a transcript overnight. Inherit one and yesterday's source set comes back, the coding matches on it, and a reading of text that no longer exists is carried forward. It therefore always runs, which costs no model calls and under a second, and it is what makes the fingerprint worth recording at all.

A sample fingerprints what each document said, not only which documents there were, so an edit moves the result's own hash and every step downstream declines to inherit. Short hashes, one per document, since the list is read rather than checked against anything.

The engine's commit is recorded on the run and gates nothing. Fixing the chunker changes what the same version produces, which is worth saying beside an older run. Making it invalidate inherited work would mean every bug fix silently re-reads every corpus, which is the expensive failure and teaches everybody to ignore the warning. So the line is drawn by consequence rather than by kind of change: would inheriting give a wrong answer? An edited document says yes. Our own code says no.

A version's name is derived in one place and stored wherever a specification is. Six hex characters of a hash of the workflow, with the declared `version` number and the project name taken out: the number so that bumping it cannot mint a version and leaving it cannot hide one, the project because it is filled in on the way into a run and would otherwise name a draft differently from the run made out of it. Whitespace at the ends of strings is ignored, the same rule as a step's comparable hash, so a YAML scalar that lost its fold is not a new version. The engine computes it and stamps it on the run and on the draft; the page reads it and hashes nothing, because a second implementation of an identity is one that drifts. Runs from before 8 September 2026 carry the number they declared instead, shown as `v3` so a derived name and a claimed one never read alike.

Models and money

The model is on the step, in its spec and so inside the spec hash. Change it and the step runs again rather than inheriting, so a cheaper model never stands on a dearer one's work without the record showing it, and one model against another becomes a comparison rather than an overwrite.

The defaults. Where a spec names none, a `code` step reads on `gemini-3.5-flash-lite` and everything else runs on `gemini-3.8-flash`. Coding is one call per chunk per document and is where nearly all the money goes; judging, writing and note-taking are each one call at the end, over material the coding has narrowed. Ruby herself runs on `claude-sonnet-4-6` unless the project setting says otherwise.

A model with no price row is refused. A cap sums a cost, so spend nobody can price reaches no allowance, and a hole in the cap is worse than a refusal naming the file to edit.

A model that is not deployed is named rather than retried. A 404 that mentions the publisher or the model means every call in the step would fail the same way, so the step stops and says which model, which models this app does run on, and what the provider said.

A failed call keeps the answer that broke it. A reply the parser could not read used to leave `Expecting ',' delimiter: line 34 column 23` and nothing else, so the one thing needed to say what was wrong with it was the one thing not kept. On 6 September 2026 that ended a run at its 214th call and took the 213 good answers with it, undiagnosable afterwards. The exception carries the string it was reading, and that string is now recorded on the failed call.

The caller is borrowed rather than rebuilt. Every provider is reached through the one entry point in `scripts/prompt_ab_test.py`, which already carries backoff and, more importantly, detection of the truncated-but-successful Vertex reply that stops mid-JSON while reporting `finishReason STOP`. That failure has corrupted a harness in this repo three separate ways, so a second caller written here would rediscover it a fourth time. The import is late and wrapped, because `scripts/` is shared ground: if the contract moves, this breaks in one place with a message saying so rather than scattering import errors through the step modules.

Concurrency. A coding step builds all its prompts and sends `concurrency` of them at once, six by default, reading the answers back by position. It is the only step that fans its prompts out. The ledger holds one connection behind a lock, so the waiting overlaps and the writing queues.

The batch lane. Half the price on spare capacity, with turnaround in hours. Declared per step and synchronous by default, because it fits a full pass over a corpus, where nothing waits on a person, and not the loop where somebody reads what came back and changes the wording. Two refusals rather than two notes. A step whose prompts are not all known in advance cannot be sent ahead, so an agentic step, whose second prompt depends on the answer to its first, is refused by name rather than having its opening turn sent off and called a saving. And the bucket stays in the EU multi-region, because Gemini 3 answers there and a job whose input sat in one country while the model answered in another would cross the boundary the whole residency arrangement exists to prevent. Answers are matched back by hashing the prompt Google echoes in its own output, so nothing depends on a custom key surviving the trip. A batch carries only the prompts nobody has answered yet, so a resubmission costs nothing twice and a collected job completes the step.

Results

What a step produces: a sample, a coding, a table, a figure, a judgement, a paragraph. You name the type yourself, in your own words.

- A source is never a result. What a sampling step produces is the sample it drew; the documents themselves stay Causal Map's.
- **A result never changes.** Run the step again and the new result sits beside the old one, so the two can be compared.
- **Every result knows which results it came from.** That chain is its lineage.

Result set. Everything a workflow's runs produced, finished or not. A run that broke at its fourth step still made three things.

Lineage

The chain by which every result records which results it came from, ending in quotes and the words in a source. If the walk back breaks anywhere, nothing else about the run counts.

You can walk back from any claim: a result says which results it came from, a figure opens the rows behind it, and clicking a quote opens its source with the words highlighted.

Quotes

A marked passage: the words and their position, checked against the real text and snapped to it where the match is exact, canonical or gapped, and kept as the model wrote it with approximate offsets, labelled as such, where the match is only fuzzy. A quote the machine cannot find in the source at all is thrown away rather than stored, because a plausible quotation that appears nowhere is the most dangerous thing an AI can produce. Everything else here exists to stay attached to these.

Match tier. How a quote was found. `exact`, `canonical` and `gapped` snap the quote to the source's own words with exact offsets; `fuzzy` keeps the model's wording with approximate offsets, labelled as such. A quote that cannot be found at all is dropped and counted, never stored.

Figures

One value a `compute` step produced, a number or a table where it grouped, carrying the unit it counted across, the filter it applied, what it took from each unit and how many units it considered, all stated rather than implied. The name is unique in the workflow and reserved names are refused.

Judgements

One verdict, produced by a `judge` step: what the material earned, why, and which quotes it rests on.

- Some verdicts are settled by arithmetic over figures, others by the AI reading the evidence against the written description. Both are legitimate, they carry different weight, and the judgement records which was which.
- **A criterion about the method cannot cite the material, so it is answered without quotations**: whether enough interviews yielded anything, whether one category swallowed the rest, whether a comparison rests on a big enough base. Keep for the reading model the criteria that need the words, such as whether the passages returned belong in the category they were given.

Evaluative judgement. A judgement that restricts a set of options which are defined and ranked by worth. It is what registration most often exists to hold, though a commissioner may register any part of the method. The three conditions are under Worth is declared by a person.

The workflow canvas

A conversation on one side and a canvas on the other.

- **A workflow is one scrolling document.** Unrun, it is the steps. Run, each result sits in the block of the step that made it. Lineage is the point of the app.
- **A margin map runs beside it**, one viewport tall and sticky, its nodes the steps and results and its edges the parentage. Click a node to scroll to its block; read a block and its node lights.
- **A step may depend on several earlier results or sources, and a result always has just one parent step.**
- **Click a node and it opens in a lightbox:** a coding step's wording and the quotes it found, a judging step's rubric and verdict. A quote opens the source around it.

Worth is declared by a person

Julian King put the objection in October 2025: AI should not be making evaluative judgements about human affairs. Quirkos and SenseMaker refuse AI analysis altogether, and f4analyse will not automate analysis of a whole corpus. The evidence-synthesis bodies draw the line in the same place, at the point where a model makes or suggests a judgement (Flemyng & {Noel-Storr 2025), and the case that the technology strengthens rather than threatens qualitative inquiry is being made alongside it (Wise et al. 2026). We agree.

- Nothing inside a scale makes it evaluative. Green is not better than red because of anything about green, and no amount of looking at the material tells you where adequate begins.
- A descriptive finding has a backstop: count the households wrong and the passages are still there. An evaluative claim has none. A standard set after seeing the numbers can be made to say anything and stay consistent with all of them.
- So the standard has to come first, from somebody who put their name to it.
- A registered workflow is versioned and never replaced, and carries who registered it and when, though on the run path that is written by the machine rather than by a person. Freezing a version, and recording whether the freeze came before the run, is designed and not yet built.
- The argument is with tools that ask a model to decide what good looks like and then present the answer as a finding.

A step is making an evaluative judgement when all three of these hold. They are properties of a declared set of levels, so they can hold on a coding column as readily as on a rubric: "code the passages where the partnership was meaningful" declares worth inside a `code` step, and a threshold set at more than half declares one inside a counting rule. The app does not test the three. It uses the structural proxy, a step of type `judge`, which is why the rubric is where it asks for a standard and why a value declared anywhere else goes unremarked.

- It **restricts** a set of options declared in advance, usually to one of them, sometimes to a band such as "adequate or better".
- Each option **says what it takes to earn it**. "Green: every partner described a change they could date and name a cause for" is a standard, because you can argue about whether the material meets it. "Green" on its own is not, because two people reading it would not agree what it means and neither would a model. A coding column whose levels are bare names is refused outright; a rubric that lists its verdicts must say what each stands for, and one that never lists them runs with the meanings reported as missing.
- The options are **graded by worth**. Health, farming and income is classification, because it ranks nothing. A frequency table over speculative, reported and evidenced has options and grading, but it reports the distribution rather than choosing a point on it.

Grading is what makes a band coherent, so `at_least` works on an ordinal and never on a category. The test applies to the answer a step gives, so a workflow is usually descriptive in the middle and evaluative at the end. Structurally: whether it ends in a judge step.

Transparency is not enough on its own. There has to be a place to disagree: override the verdict, rewrite the rationale, reject a cited quote, mark a criterion unanswerable. The override would sit beside the machine's verdict so the difference is itself a finding, carry who and when, and be what everything downstream used. None of it is built. Today the only human decision the app records is accepting or withdrawing a whole result.

Every claim points at a quote

- Any narrative or verdict about the material must lead back to the quotes it rests on (Moravcsik 2014). The system refuses to store a note or a judgement that cites none, and it gets there by requiring the coding to be a declared input of the step making the claim, so a claim cannot drift away from its evidence. A table is not held to this.
- This is the likeliest way a tool like this fails, and it fails without showing: two steps from the source the prose still reads well and the numbers still look precise, while nothing connects either to a document. Note it's easy for an AI tool to reach a conclusion and then search the text for quotes to back that up. Rubicon does not fall into that trap.
- **Some findings are about the method rather than the material:** was contrary evidence looked for, was the sample drawn in a way that supports the claim. No quotation can evidence the absence of a search, so those cite the run record instead, and the judgement says which kind each verdict was.
- A criterion should be one kind or the other, and nothing refuses one that is both. "Was anything found that cuts against the hypothesis, and how was it handled?" is two questions, and asked as one it makes a model rate the method by reading the evidence.
- A band should stand for one state of affairs, and this one is on you: nothing reads a band's description. Where the worst band covers both "contrary evidence was found and it is damaging" and "nobody looked", the reader cannot tell which happened.

A negative finding is the hardest case, and the test is what the claim CITED rather than what it was handed. Asking which of a theory's links the material never mentioned hands the step every quote the coding found, and the answer is about the links with none. So a note that cites nothing rests on the search instead: what was looked for, over what frame, how much was read, and how many rows it found.

A proportion states its base, and the base includes the documents that gave nothing. Build a denominator from the passages already coded and a document that said nothing disappears, so "of the sources that gave me something" reads as "of the sources". On a sparse question, which unintended harm usually is, that inflates the headline figure where it matters most.

What the machinery reports, beside what it found, is the passages dropped because the quote could not be located, how much of each document was examined, and how many model calls there were and what they cost.

Build the thing, and not only the verdict

A verdict is one word standing for eight links, three hundred passages and nineteen documents, and tracing back through it does not rebuild what was compressed. So where you would expect a graph, a table of outcomes or a cross-tab to be in front of you, the workflow builds it and declares it as a result, which usually costs nothing because it was a step on the way anyway. Such a workflow can stop there: a table can be accepted as the answer as readily as a verdict. The mirror fault costs more, so a question that wants a number should not pay for a graph nobody asked for.

Say what you are comparing against

A programme did well: compared with what? Seven answers, and they are not interchangeable.

- **A denominator.** Eleven of the fourteen partners. Needs a unit and a stated base.
- **An alternative.** Which of two readings the material better supports. No denominator at all.
- **A written standard.** A rubric agreed in advance. The only one that works when there is nothing to compare with.
- **A comparison case.** The programme before it started, a similar one elsewhere, a site that got no support. Rests on the cases being alike in the ways that matter.
- **What was expected.** The theory of change, the target, the plan. Cheap, and it measures delivery rather than worth.
- **What would have happened anyway.** Usually the hardest. Approachable through causal mapping and other generative accounts of causation, which treat people as causality detectors whose judgements carry counterfactual implications.
- **The evidence itself.** Sometimes the finding is that nothing here is well enough evidenced for any of the above.

Most confusion in evaluation reporting comes from sliding between them: counting how many people said something, then writing as though that settled whether it was any good.

Search both sides equally. Iterating on what you have read is how the work gets better, and it is also how a method comes to fit one corpus and nothing else. You read the transcripts, the coding instruction gets narrowed until it finds what you saw, the threshold settles where the numbers happened to fall, and the report presents the result as though the method had been decided independently of it. Nobody has deliberately cheated, and the reader cannot see it either way.

Searching both sides in one pass over the same text is a convention rather than something the app enforces. What the app does is check, where two codings are compared, that they had the same opportunity, per align. It reports the disparity and refuses nothing.

Hold material back. Develop the workflow on a subset, or on analogue material from a previous year or a similar programme, then run the settled workflow over the rest. Where the workflow writes its own tests, split the corpus so that the half that wrote a test never scores it.

Say what you did not read

A workflow may read less than the whole corpus, by drawing a sample of documents or by narrowing to the passages about a subject. Both are ordinary. What is not ordinary is reporting the result as though everything had been read. A number's frame is whatever reached the step that produced it, so a draw makes the frame the documents drawn and a narrowing makes it the passages retrieved. A figure that leaves its frame out says something other than what it means.

Narrowing is right where the question is about a subject the material names in its own words, and where reading everything costs more than the answer is worth. Two cases where it is wrong. A prevalence over people or documents needs every document opened, since a document that was never retrieved is not a document that had nothing to say. And a comparison between causes is only as fair as the query: retrieve on the outcome and every cause competes on the same ground, retrieve on one of the causes and the comparison was settled before the reading started.

An absence survives narrowing where it is stated as an absence in the search. What was looked for, how close a match was required, and which documents contributed nothing are all on the asset. "Nobody mentioned childcare" is a claim about the corpus and needs the corpus. "No passage above 0.45 on childcare, in any of the thirty-seven interviews searched" is a claim about the search, and it is the one the run can support.

What it refuses, and what stays soft

- **"Read the documents and tell me whether the project was effective."** Answered in one pass, that produces the thing this app exists to replace: a confident verdict, no stated standard, and no way for anyone to disagree with it in particular.
- **"Is this a good project?"** is four questions in one coat: good for whom, compared with what, against whose standard, and how much of that this material can answer.
- The test is whether the question is well defined. Nothing is refused for being about tone or feelings or anything else that sounds soft.
- **Refusing is rarer than it sounds.** "What percentage of the budget went on training?" is well defined where a report states the figure and unanswerable where none does, which is a fact about the corpus. The answer is usually a workflow that goes and looks. If nobody wrote the figure down, that is the finding, with the search behind it.
- **Some questions are well defined in parts and not as a whole.** "What share of the fellows who wanted a policy meeting got one?" is two codings and a ratio. A single pass would be arithmetic done in a model's head.
- **A question whose answer is in a spreadsheet is refused, and that is the point.** Hours between landfall and the first cash payment, cost per person reached: those live in a payments system with no words behind them. Everything here rests on a number being followable back to the words somebody said, and mixing the two makes every number only as traceable as the weakest. The workflow says which part of the question it answers, and you bring the other figures in when you write.
- **A number spoken in a document is in scope.** "We reached about four hundred children", said by a school director, has words behind it, a speaker and a position in a source. A coding step declares a numeric column and the counting does arithmetic on those.
- **A judgement may refuse the question.** Where the material shows the question rests on something that did not happen, or compares two things the sources cannot compare fairly, the judge says so instead of awarding a band, and cites what that rests on. Awarding a band is worse, because the band gets quoted and the doubt does not.

Named methods are recipes

No named method reaches a code path. Process tracing, contribution analysis, realist evaluation and outcome harvesting are recipes written in the general parts, and Ruby offers one when a question suits it. A tool built round a fixed list is wrong the first time somebody wants something slightly different, which is most of the time, and a method encoded in software is a method nobody can argue with.

The recipes, in one line each.

- **Process tracing.** Rival published theories against a corpus, using the four evidence tests (Ricks & Liu 2018), with the tests written before the material is read and a synthetic set used to check they can tell the theories apart at all.
- **Realist evaluation.** No theory to start with, generating candidate mechanisms, then measuring how many of them a competent reader could have written without doing any fieldwork.
- **Contribution analysis.** The theory of change the programme committed to, each link tested, and the rival explanations for the whole outcome taken as seriously as the programme's own account. The product is an annotated theory of change rather than a verdict.
- **Outcome harvesting.** Mostly a conversation between people, and it should stay one ({Wilson-Grau 2018). Two analysis questions in the middle are worth a machine: whether the outcomes form a pattern of progress against the objectives, and which objectives have nothing against them.

Where a method wants something that would be useful more widely, it becomes a general part rather than a special case. Outcome harvesting needs the thing being counted to be a person rather than a document, and so does anything else about actors, so that is a dial. The test is whether a second method would use it, and the ones that pass keep arriving from three directions at once.

Where this sits in the literature

- Scriven's logic of evaluation separates description, "What's so?", from valuing, "So what?", then reaches a verdict through criteria of merit, standards of merit and a synthesis into overall worth (Scriven 1980, 1991; Fournier 1995).
- Davidson puts it as a test: an evaluation must ask not only "What were the results?" but "How good were the results?" Her worked failure is an executive summary of counts and percentages with no verdict in it (Davidson 2014).
- Gargani and King's second principle says the same about worth: "an evaluator cannot ascertain the value of an impact solely by studying it because value is not located within it". The philosophy stays open, and Putnam's argument against the fact and value dichotomy is the standing objection.

Open questions

An item leaves when it is decided: built and durable it graduates into the principles above, abandoned it goes without ceremony.

- **Validating a workflow on synthetic contrasting texts.** The `synthesise` step is built with its refusals; what is missing is the last step, the one that compares the verdicts awarded against the verdicts expected and reports the separation as a figure. Form: a `synthesise` step, then the ordinary workflow run over what it produced, never a path of its own. Design in `rubicon/validate a workflow on synthetic contrasting texts xkTODO xpSTEVE.md`.
- **Near-miss reuse.** Identical reuse is settled by hashing the request. What an evaluator meets is a theory of change very like one this project already holds, where rebuilding costs an hour and ten dollars. Two things would make it safe, neither automatic: the difference stated in terms somebody can judge, which links differ and which clause changed, rather than as a similarity score; and the reuse agreed rather than assumed.
- **Reading an accepted answer rather than a raw asset.** `from_run` reaches an asset, which is the raw output of a step and not what anybody agreed to. So a comparison can be built on a run that was never accepted, or one whose acceptance was later withdrawn, and nothing in it says so. An input of the shape `{answer_to: <question id>}` would resolve through the answers instead, refused where the question has none rather than falling back to whatever ran last. Design in `rubicon/xkTODO a comparison should read an accepted answer xpCLAUDE.md`.
- **Building a theory of change, rather than checking one.** Checking is tractable, since each link is a proposition and asking the material what supports it is close to what the app does now. Building is harder: the object is assembled from hundreds of fragments and no fragment contains it, so the categories cannot all be declared in advance and a source read late can change what an earlier passage meant. Causal Map does this well, so the question is what an auditable version looks like when the structure emerges rather than arriving.
- **Running a step at a time from the page.** The engine has `until`, `continue_from` and `rerun`, and the command line reaches them. The page posts a draft and gets back the whole workflow, so the way this app is meant to be used is the way nobody using it can work. One naming trap waits there: `runs.status` already uses `partial` for a run whose steps failed, so the workflow-level word has to be a different one.
- **Whether the sample belongs to the workflow or to the run.** Run over twelve sources, then over all fifty-three: intuitively the same workflow twice. The vocabulary says two versions of one workflow, since the sample decides what the answer should be. What is unsettled is only what the pane shows, which needs a run picker either way.
- **Forking a workflow from a step**, discarding everything downstream. That makes a new workflow. Not built.
- **A place to disagree.** Overriding a verdict, rewriting a rationale, rejecting a cited quote, recorded beside the machine's version and carried downstream. The principle is above and the surface does not exist.
- **A composite as a result in its own right.** `composes` assembles one, and it rides on the coding's asset, so it cannot be accepted as an answer or drawn. Whether it becomes a type of its own is the open decision.
- **Comparison against what was expected.** A workflow can compare two codings and compute a proportion. It cannot yet read a written expectation and ask the material whether it happened, which would report which links nobody mentioned, something an evaluator cannot get any other way.
- **The absent-link register.** A proposition the material never mentioned reads as a nought in a table, and the absence an evaluator most wants has nowhere better to live.
- **Iterating a step until a figure passes a threshold.** A bounded loop over one step, which would make the coverage dial settle itself. Left late because a loop makes a run non-deterministic in a way a seed cannot fix, so registration has to say what a version means when the number of turns is decided at run time.
- **The draft-to-run link.** A run stores a copy of the workflow it ran rather than the id of the draft it came from, so the ledger cannot say which proposal an attempt came from. Two things bridge it by the workflow's id and the version's derived name. What it still cannot do is tell two drafts of one recipe apart. One column, written when the page posts the draft's id, would close it properly.
- **Project home shows raw column names**, fifteen lines of `s_#Name of province` and the rest, where the reader should see the name a person would use.
- **Flagging a background document has no button.** You type the value into a custom column on the source.

- **Arranging the canvas by asking.** "Show me the quotes, longest first", "put the two codings side by side". Ruby proposes workflows and has no tool that touches the canvas, so today the page shows what it shows. Not built.

Not settled at all:

- Making the unit a person or an organisation rather than a document, when who they are is only discovered during the coding.
- Expressing a base that is not a set of documents, such as "the forty partner organisations", when only twelve appear in the material.
- Pausing a workflow for people to do something in the world, then picking up where it left off.
- Merging results that turn out to describe the same thing, keeping the trail back to every passage that contributed.
- Naming.

References

Flemyng, & {Noel-Storr (2025). *Position Statement on Artificial Intelligence (AI) Use in Evidence Synthesis across Cochrane, the Campbell Collaboration, JBI and the Collaboration for Environmental Evidence 2025*. Cochrane Database of Systematic Reviews. https://doi.org/10.1002/14651858.ED000178.

Moravcsik (2014). *Transparency: The Revolution in Qualitative Research*. APSCC Newsl., *47*, 48--53. https://doi.org/10.1017/S1049096513001789.

Ricks, & Liu (2018). *Process-Tracing Research Designs: A Practical Guide*. PS - Political Science and Politics, *51*, 842--846. https://doi.org/10.1017/S1049096518000975.

Wise, Gresalfi, & {Spencer-Smith (2026). *Why AI Is Not the Enemy: Opportunities to Strengthen Core Commitments of Qualitative Inquiry Through Trustworthy AI-in-the-Loop Analysis*. International Journal of Qualitative Methods, *25*, 16094069261435579. https://doi.org/10.1177/16094069261435579.

{Wilson-Grau (2018). *Outcome Harvesting: Principles, Steps, and Evaluation Applications*. IAP.