

— PROCESS AUTOMATION

Data Preparation for a Process Automation Foundation Model

Deriving a multi-task training corpus from **screen recordings of human work**

AUTHOR

Neurologic AI

PUBLISHED

July 2026

REFERENCE PROCESS

Financial back office

ABSTRACT

The only complete record of how a back-office process is really executed is the operator's own screen. This paper describes a data preparation methodology that turns raw screen recordings (screenshots paired with user-interface events) into a supervised, multi-task training corpus for post-training a vision-language model into a process automation foundation model. The methodology assumes only that work arrives as discrete cases, that each case is executed as an ordered sequence

of interface actions, and that a written description of the process exists or can be produced. We describe seven preparation stages, in dependency order: per-step one-line summarisation; case boundary detection, which reads those summaries to work out which steps belong to which case; iterative refinement of the written process guide against the detected cases; and then four generation stages built on both: next-action generation with chain-of-thought, goal-based supervision, step-based question answering grounded in the current screen, and process-knowledge question answering written against the guide. One principle accounts for most of the engineering: every stage is generated with more context than the trained model will have at inference, and that surplus is then removed from the training prompt by automated gates that abort the build on violation, so the model learns the process from labels rather than from instructions. We report the composition of the resulting corpus, the case-level split that prevents adjacent-step leakage, per-stage verification results, and the residual defect rates measured rather than assumed.

7

Preparation stages, in dependency order

4

Training tasks in one uniform format

80/10/10

Split, assigned once per case

Abort

The build, on any leakage-gate violation

— 01

Introduction

Conventional process automation encodes a documented procedure as rules or selectors against specific screens. The result covers one process, and the documentation it is built from describes the intended path rather than what operators do.

A foundation-model approach learns the process instead. A single vision-language model is post-trained on recorded work so that it acquires the sequence, the decision gates, the vocabulary, the interfaces and the recovery paths as internalised knowledge, and can then

be asked what happens next, what the current activity is trying to achieve, why a step is necessary, where a control is on screen, or how the process handles a given exception, all from the same weights.

The bottleneck is data. A recording of an operator working a case is unlabelled at the semantic level: an event records that a click occurred at a location, not that it selected a particular record for a particular reason. It is also unsegmented, both across cases and within a stream. This paper covers how that raw signal becomes a multi-task supervised corpus; it does not cover model architecture, training schedule or downstream evaluation.

The methodology is described in process-neutral terms. It has been applied to a document-heavy financial back-office process in which operators work cases across a case-management system, a line-of-business ERP, a spreadsheet register, a word processor and an email client. Corpus volumes are deliberately not reported. The measurements that are (defect rates, leak rates, token budgets and the thresholds they set) are given because they are what justifies a design decision rather than what describes the deployment.

1.1 Pipeline order

The stages form a chain, and the order is a dependency order rather than a presentation one. Summarisation runs first, on the raw capture. Boundary detection reads those summaries (not the images) to decide which steps belong to which case and where each case starts and ends. The written process guide is then refined against the cases that came out of it. Only then do the four generation stages run, each conditioning on the case files, the summaries and the refined guide. Every arrow in that chain is also an invalidation arrow: re-running boundary detection changes the case identifiers and the step numbering, so the goal, step-based, chain-of-thought and assembly stages must all be rebuilt behind it.

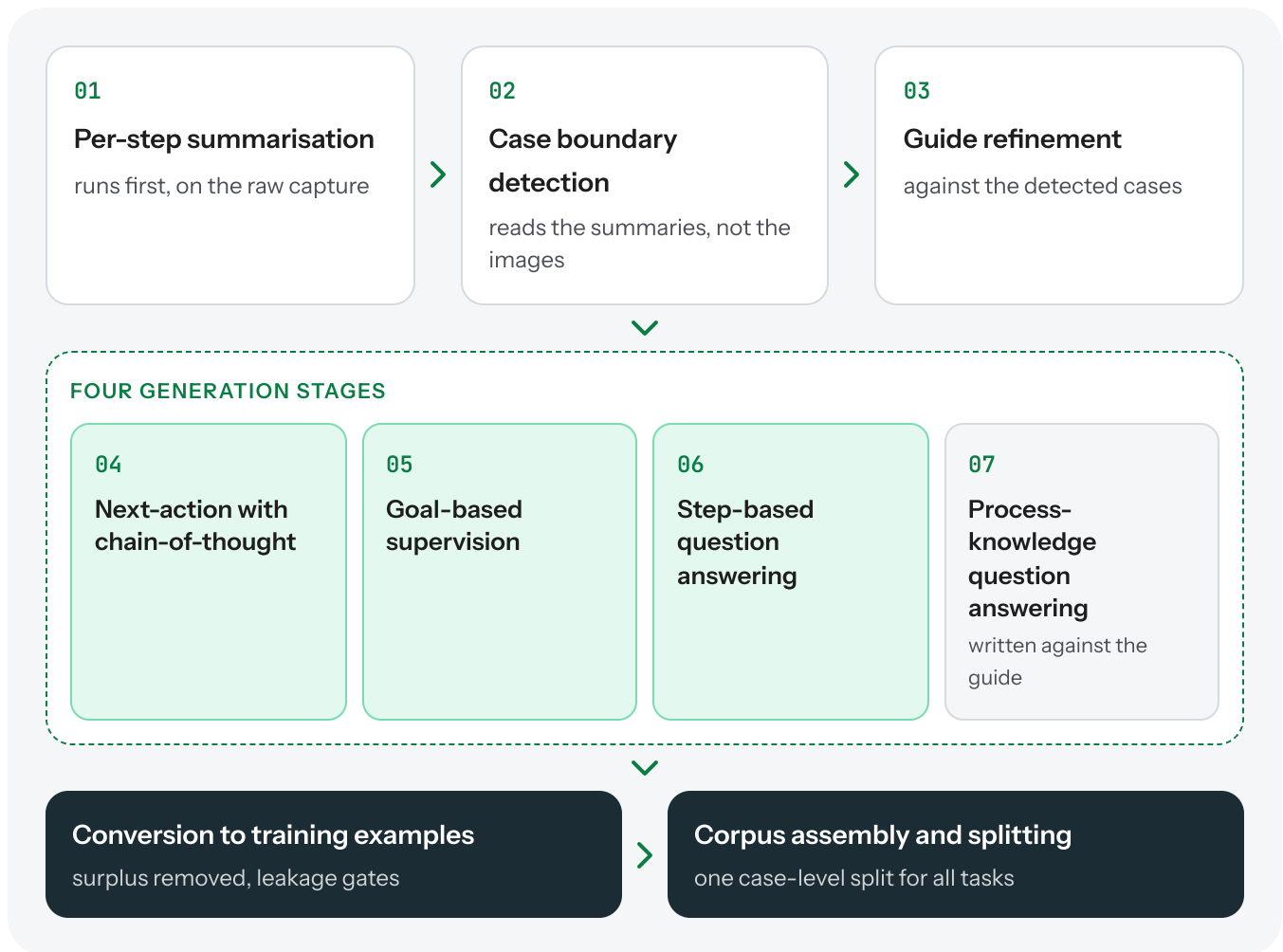


Figure 1. The stages in dependency order. Every arrow in the chain is also an invalidation arrow.

1.2 Design principles

Four principles recur and explain most of the decisions that follow.

01 Generate with surplus context; train without it, and enforce the removal mechanically.

Each generation stage receives inputs a deployed model cannot have: a lookahead over future actions, the ground-truth next action, the written guide, a category assignment. Every one is removed when the record becomes a training example, and each conversion stage asserts the absence of a specific token list; one also checks for any shared eight-word run with the guide. A violation aborts the build.

02 The case is the unit of everything.

Context windows, concurrency, resumption, provenance and splitting are all keyed on the case. Adjacent steps of one case share most of their context, so splitting by example would place near-identical examples on both sides of a partition.

03 Every record carries its provenance.

Each generated record and each training example resolves back to its source screenshot and interface event through an `origin` tuple of operator, date, source step index and timestamp.

04 Measure the defects.

Every stage reports what it dropped, what it repaired, and at what rate. Where something was not verified, Section 12 says so rather than letting silence imply a check.

AT GENERATION: SURPLUS CONTEXT

A lookahead over future actions

The ground-truth next action

The written guide

A category assignment



AUTOMATED GATES

Forbidden-token list per task, plus an eight-word shingle check against the guide

Violation aborts the build



TRAINING EXAMPLE

Every surplus input removed; the model learns the process from labels rather than from instructions

Figure 2. Generate with surplus context; train without it, and enforce the removal mechanically.

The recording substrate and the summary layer

2.1 Raw recordings

For each operator and working day the recorder emits an ordered sequence of *steps*. Each step holds a screenshot of the display at the moment of an action and an event file describing that action. Extracted on-screen text is also captured but is usually empty, and is retained only where populated.

A single operator-day stream interleaves work on several cases with work belonging to no case, and a single case is spread across several days and sometimes several operators. Case lengths vary by more than an order of magnitude, which has consequences for concurrency and for splitting that recur throughout this paper.

2.2 Stage: per-step one-line summarisation

The first stage converts each raw step into a compact semantic record. A vision-language model is shown the step's screenshot and its interface event, together with the process guide and the summaries of the preceding steps in that stream, and emits one JSON object: a screen identifier for the application and screen in view; the action as `{type, target, value}`, where the target is a human-readable description of the control rather than a coordinate; the business identifiers visible on screen, including those belonging to other cases visible in a list view; a one-sentence summary of what the operator did; and the extracted entities as a flat key-value map.

ONE JSON OBJECT PER STEP

Screen
identifier

Action as
`{type,
target,
value}`

Business
identifiers on
screen

One-sentence
summary

Extracted
entities, flat
key-value map

This layer is the substrate for the whole corpus: every later stage reads the summary and the extracted entities rather than the raw screenshots when it needs history, which is what makes twenty-step context windows affordable.

Three implementation details matter. **Steps within a stream are generated sequentially by design:** each step's prompt includes the preceding summaries, so parallelism is taken across operator-day streams, not within them. **Runs are resumable and versioned:** output is written under an attempt number, a step whose output exists is skipped, and a new run bumps the attempt rather than overwriting; the attempt that produced the data is what every downstream stage reads. **Raw responses and exact request messages are retained** alongside each record, with image payloads stripped, so any record can be audited against what was asked.

03

Stage: case boundary detection

This stage works out which recorded steps belong to which case. It is not a preprocessing convenience: it is a generation stage in its own right, with its own model calls, its own labels (case attribution, branch verdict, stages observed, completeness) and its own verification, and it is what makes every later stage possible. An operator interleaves many cases inside one working day, and one case is worked across several days and several operators; this stage untangles that.

It reads the one-line summaries of Section 2.2 (not the raw capture) using each step's summary, resolved identifiers, screen identifier and the upstream stage-position hint. From each step's record it takes the identifiers in a fixed order (case, order, memo, invoice, customer, and any other case identifiers visible) and drops the fields it does not need to re-audit.

3.1 Four steps, four outputs

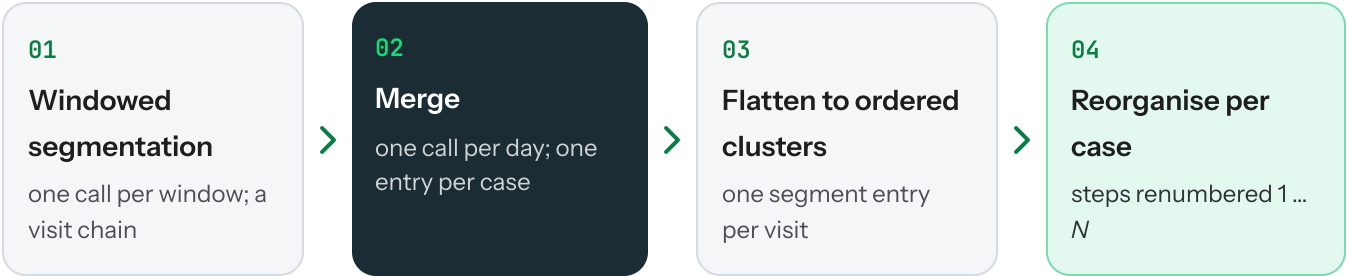


Figure 3. Case boundary detection: four steps, four outputs.

1. **Windowed segmentation**, one model call per window. Each window is segmented independently into a *visit chain*: contiguous, non-overlapping visits, each labelled with a case identifier or with a no-case marker.
2. **Merge**, one model call per day. It takes the day's visits (all its windows concatenated) plus the guide, and returns one entry per case: same-case visits grouped, bridging no-case runs absorbed, and clearly wrong attributions corrected. It also judges the case's *branch* and the workflow stages it observed.
3. **Flatten to ordered clusters**. Each disjoint step range becomes its own segment entry, so a case revisited later in the day gets a separate entry per visit, which is what stops a lookback clamp reaching back past an unrelated earlier visit to the same case. Gaps are labelled as no-case.
4. **Reorganise per case**, renumbering steps 1 ... N in true chronological order.

The three later formats are consumed by different things: the flattened clusters feed the day-centric stages of an earlier pipeline generation, and the per-case files feed every stage in this paper.

3.2 Windowing, forced by context

Days are long. At a measured 75.6 tokens per compacted step, a whole day's input runs from tens of thousands of tokens to several hundred thousand for the longest streams. A whole-day call would also put hundreds of thousands of tokens of mostly irrelevant context around each decision, so days are split into non-overlapping windows.

The window size was set by measurement rather than estimate. Over eight 100-step windows against a 32,768-token context, input ran 8,579 to 13,567 tokens and output 3,545 to 17,923: worst case 94% of the entire context, and one of the eight exceeded the cap and failed outright. Output is both the dominant and the volatile term, because it scales with how many visits and cases a window happens to contain, varying fivefold for the same step count. At 75 steps the worst case is around 24,600 tokens; 150 and 200 exceed the context before generation begins, so they are not reachable at that context length regardless of how the output cap is set. The cost of non-overlapping windows is that a case spanning a boundary is reported independently in two windows with nothing joining them, which is precisely what the merge exists to repair.

75.6

tokens per compacted
step

32,768

token context

94%

of the context, worst
case at 100 steps

~24,600

tokens, worst case at
75 steps

Internal reasoning is enabled, and here too the decision was measured. Disabling it made calls three to five times faster and cleared 456 easy windows at 36 per minute, but 42 dense-day windows then failed all three attempts, unable to produce a valid gapless visit chain without a reasoning step. Thinking off is adequate for sparse days and required for heavily interleaved ones.

3.3 Coverage as a hard gate

A segmentation that silently drops steps is worse than a visible failure, so the visit chain is checked by forward scan: the first visit must start at the day's first step, each visit must start where the previous one ended plus one, and the last must end at the day's last step. Gaps, overlaps and out-of-range visits are reported individually, and a chain that does not account for every input step exactly once is rejected and retried. The merge is held to the same standard.

3.4 Merging a long day, and where the model stops counting

The merge needs every visit of a day in one view, because grouping a case is a whole-day judgement. Its input therefore grows with the day: a measured 0.167 visits per step at roughly 38 tokens per rendered visit, so a long day's visit list alone exceeds the context. Two mitigations apply, in order:

- **Visit summaries are truncated word-wise** until the rendered input fits, down to a floor. Case identifiers and spans are never touched; only the prose gets shorter, and only on the few days that need it.
- **Above a visit threshold the day is chunked**, each chunk merged independently and the results unioned in Python by exact case-identifier match. Chunks keep *global* visit indices, so nothing needs remapping afterwards.

The threshold exists because the failure above it is not a context problem. Measured on the corpus, a 391-visit day hit the output cap outright, but a 143-visit day completed cleanly and *duplicated a visit index*: the model simply miscounts across many indices. Raising the output cap or the context fixes only the first kind of failure; no amount of either makes the model

count 300 indices reliably. The price of chunking is that a no-case visit sitting exactly on a chunk boundary cannot be tested for bridging, because its neighbours are in different calls; those stay no-case, which is the conservative direction.

A related division of labour runs through the whole merge: the model returns *visit indices* rather than step ranges, and the ranges are computed from them. The model cannot then produce a range that disagrees with the visits it grouped, and the output is roughly half the size, which is what lets the branch and stage judgements fit in the same call.

3.5 Completeness is derived in code, not asked for

Each merged case carries a *branch* (the positive or negative side of the process's main decision fork: billable or non-billable in the reference process, or undetermined), the workflow stages the model says it observed, and a *progression*: fully processed, half processed or minimal.

Progression is **derived in Python from the observed stages, not taken from the model's own verdict**. The split is deliberate and follows what the model is reliably good at: it can answer "did I see charge entry on this case", and it cannot hold a six-stage checklist and report completeness consistently. So the model reports evidence and the code applies the rule.

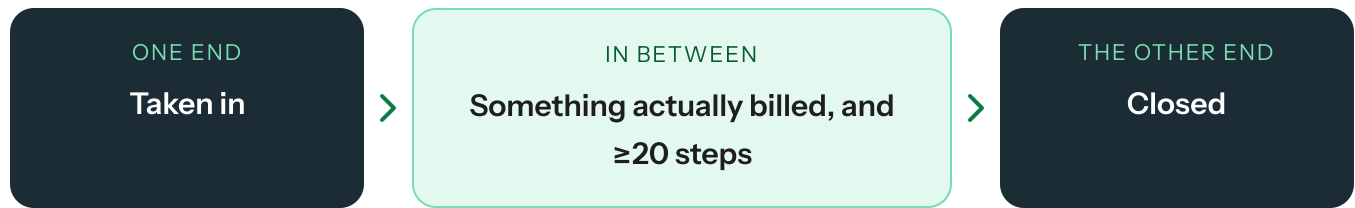
The rule itself was arrived at by measurement, and the iterations are worth reporting because the obvious rule was badly wrong.

Rule	Complete	Share	
Every required stage observed	21	2.2%	
Case-closed stage alone	234	25.1%	
Intake + closed, no billing evidence required	52	5.6%	
Intake + closed + a billing artefact + ≥20 steps	51	5.5%	

Table 1. Completeness rules tried against the positive-branch cases, and how many each admitted.

"Every stage observed" scored 2.2%, and among its rejects were cases that demonstrably finished: one showing the ERP's billing-document screen on a verified screenshot, another running hundreds of steps across almost the whole spine, both failed purely because no filing step happened to be captured. The corrected rule reads the case as a **trail**: taken in at

one end, closed at the other, and something actually billed in between. *Intermediate gaps are read as capture gaps, not process gaps*: a case cannot be closed on the positive branch without having been priced and charged, so a missing pricing stage means the recorder missed it.



Both ends are required because each fails alone. The closing stage by itself admits a three-step case that only ever displayed a closure tab; spot-checks found a merely *viewed* closure tab being reported as a close, making it the least reliable stage in the corpus. Intake by itself says nothing about whether the work finished. The billing artefact and the size floor sit in between so the trail cannot be satisfied by opening a case and closing it with nothing done. Two further rules fall out of the process: a contract type that legitimately skips the decision gate has the preceding gate substitute for it, and the negative branch needs only its verdict and its close, because that branch is two stages and both are well evidenced.

Interestingly, **intake is the binding constraint, not the billing evidence**: only 8.4% of positive-branch cases have intake captured at all, but 52 of those 78 also reach the closing stage; when the recording catches the start of a case, it usually catches the finish too.

The cases that pass this rule are the corpus: the full-trail cases every later stage reads.

3.6 Ordering on the real clock

Step numbers are a counter local to one (operator, date) pair, so they cannot order a case that spans several. Every step's event file carries a timestamp to the millisecond on a clock shared across operators (verified as a zero offset over eight operator-days), so ordering is a plain numeric sort with no heuristics and no stage-label guessing; identical timestamps fall back to (operator, source step), also deterministic.

Ordering deliberately **does not use** the stage-position tags. They are free text with dozens of variants, around one in eight is unreliable, and some carry two stages at once. More fundamentally, real chronological stage paths are *not monotonic* (operators loop back constantly), so sorting by stage would fabricate a sequence the recording contradicts and would erase the rework signal entirely. Stage tags stay a label on a step, never a sort key.

Within a case, steps are grouped into **clusters**: maximal contiguous runs of the same (operator, date) after the global time sort. Clusters are deliberately not the same as per-case sessions, because same-day operators genuinely interleave: one reference case with two sessions produces seven clusters, two operators ping-ponging six times across a single day. Ordering whole sessions as units would silently merge those handoffs and invent a sequence that never happened.

The output is a **pure ordering index**. No content is copied: screen, action, summary, stage, identifiers and the screenshot all stay in the source tree, reachable from (operator, date, source step) through path templates written into each case file. The one non-derivable field kept is the timestamp, so the ordering stays auditable without re-reading the capture.

One consequence of the attribution looks like a double count and is not: the per-case step counts sum to slightly more than the number of distinct steps, because a few steps are claimed by two cases at once (an operator advancing several cases from a single worklist screen). Those steps appear in both case files, which is correct.

Verification is structural: within each (operator, date), source step numbers must never invert across that operator's own clusters. Nothing enforces this; it falls out of the clock agreeing with the step numbering, so a violation means the input is inconsistent, and the run reports it.

3.7 What the case files mean downstream

The renumbering is the most important structural fact about the corpus. A case file numbers its steps 1 ... N contiguously, but the underlying source steps are not: a case may own one isolated source step, then a short run, then a much longer run much later, across dates and operators, because the operator worked other cases in between. Consequently:

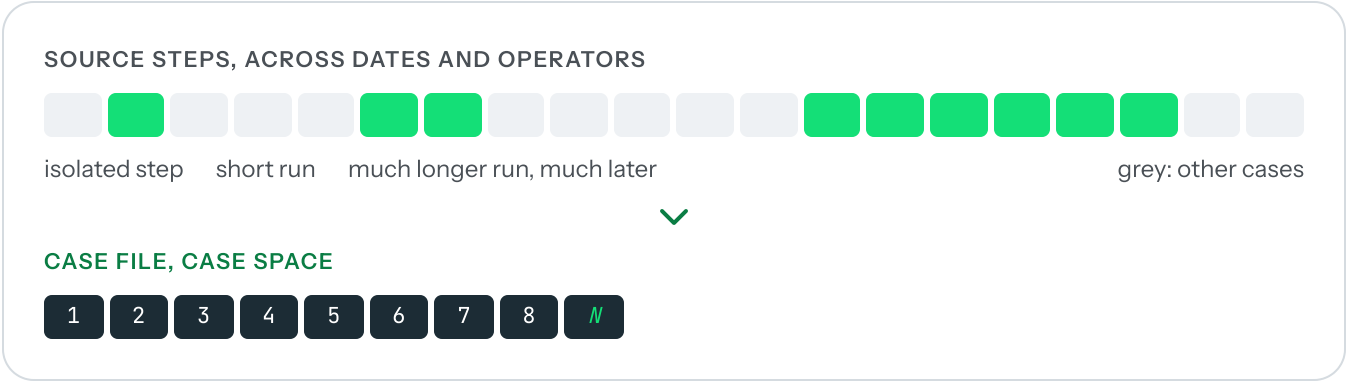


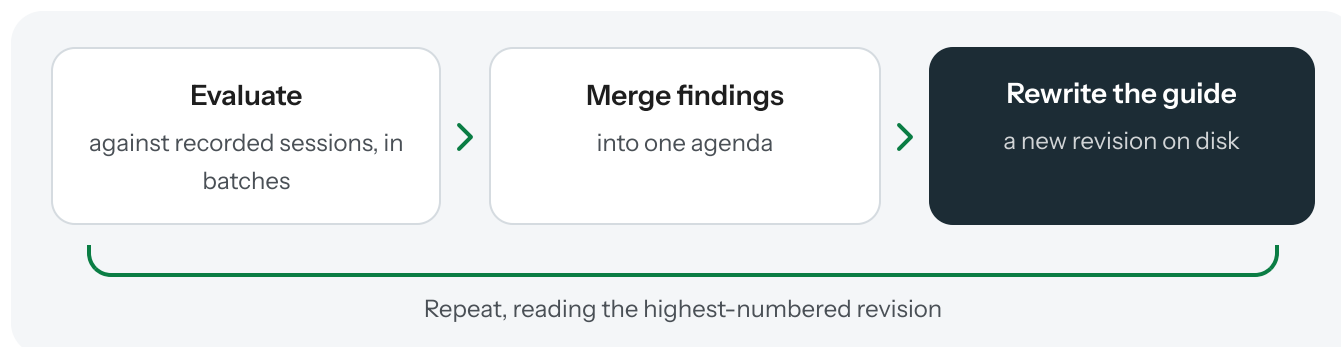
Figure 4. A case file numbers its steps 1 ... N contiguously, but the underlying source steps are not contiguous.

- **All context windows are taken in case space.** A twenty-step lookback from case step 30 means case steps 10 to 29: contiguous, never crossing into another case's history.
- **Each step in a window is resolved individually.** Every step carries its own `{operator, date, source step}` triple, and the case file carries path *templates* rather than paths, for the summary, screenshot, event and screen text. A window spanning a jump in source step numbering, or from one operator to another, is assembled without a seam. Because the templates are read from the case file, relocating the recording tree requires no change to any generation stage.
- **Clamping to the case is free.** Every step in a case file already belongs to that case, so clamping a lookback to $[1, N]$ is clamping to the case boundary. An earlier iteration of this pipeline worked on raw streams and needed an explicit cluster-and-gap model to stop one case's history bleeding into another; the case-centric layout eliminates that machinery.

— 04

The written process guide and its refinement

Three generation stages condition on a written description of the process and a fourth is written entirely against it, so its quality sets a ceiling on the corpus. We refine it against the recordings themselves, in a loop: evaluate the current guide against real recorded sessions (drawn from the cases boundary detection produced, balanced across the two branches of the main decision fork), merge the findings into one agenda, rewrite the guide against that agenda, and repeat, each iteration reading the highest-numbered revision on disk so an interrupted run resumes.



Batching is forced by context: the selected sessions together exceed the model's context, so they are split into batches sized such that guide plus evaluation prompt plus batch fits the input budget remaining after the completion reservation and a safety margin. Each batch produces its own evaluation, and the batch evaluations are merged before the rewrite.

In the reference deployment the loop was run to a fourth revision, yielding a ~43 KB, 440-line document covering nine spine stages with substages, five variant flows, ten decision gates, pricing-authority precedence, evidence rules, invoicing and consolidation, filing and dispatch, fourteen error-handling entries, five blockers, a ten-row identifier map, an application-specific appendix and nine cross-cutting notes.

~43 KB

440-line document,
fourth revision

9

spine stages with
substages

10

decision gates; five
variant flows

14

error-handling entries;
five blockers

Two operational rules matter. **Promotion is deliberate, never automatic**: running the loop produces a *candidate*, and pointing the generation stages at a new revision is a separate explicit act, because changing the guide invalidates every record generated against the old one. **The guide's section numbering is not part of the process**: the document heads its sections "Step 4", "6.3" and so on, but a model that learns "Step 4" cannot answer an operator who says "the billability check". A mapping from every numbered heading to a descriptive stage name is maintained separately and shared by every generation prompt in the corpus, and emitting a bare stage number is treated as a defect and checked at build time. This is why four independent generators produce one vocabulary.

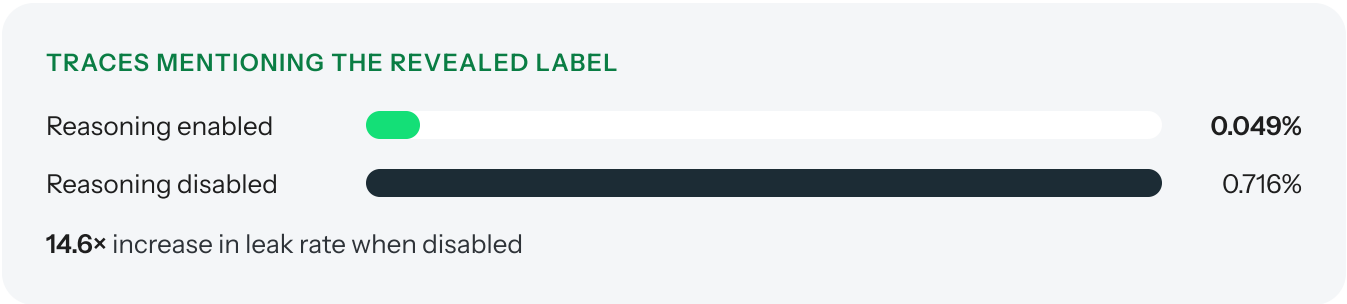
— 05

Stage: next-action generation with chain-of-thought

Supervision for the next action is free (the recording contains it) but supervision for the reasoning that justifies it is not. This stage synthesises the reasoning. For each step of each case the generator receives the previous up-to-20 one-line summaries in case space, the current screenshot resized and embedded, and **the step's own ground-truth summary**. It must produce a reasoning trace explaining why that action plausibly follows from the history and the screen, the indices of the previous steps it relied on, and a confidence score.

It operates in *no-leak mode*: the ground-truth action is revealed for internal alignment only, and the trace must never mention it. This revelation is what makes the stage chain-of-thought generation rather than question generation.

Internal reasoning is left enabled, decided by measurement. An A/B test on this exact task measured a 14.6× increase in leak rate (from 0.049% to 0.716% of traces mentioning the revealed label) when it was disabled.



Steps here have *no cross-step dependency*: each reads the pre-existing summary layer and its own screenshot, never this stage's own output, so the queue unit is a single (case, step) pair and every step may run concurrently. Because nothing reads this stage's output, a resumed run produces exactly what a from-scratch run would. Requests are streamed, because the proxy in front of the inference server terminates connections silent for roughly 100 seconds; retries use exponential backoff. Two defensive details were ported forward from an earlier production run rather than left to recur: tolerating prose before a fenced JSON block, and clamping the reported evidence indices to integers within the case's range.

06

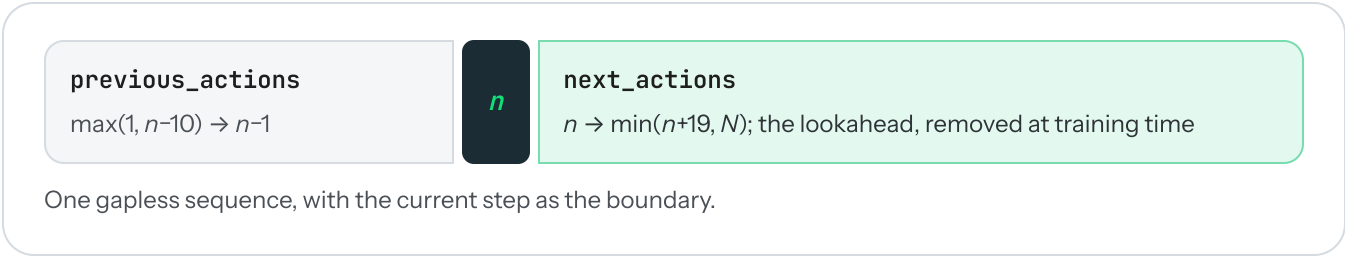
Stage: goal-based generation

This stage supplies purpose: for each step, the goal that step's action serves and the actions that goal still requires. The generator sees the summaries of step *n* onward (a lookahead of up to twenty future actions) and infers backwards what goal they accomplish. Because the first lookahead entry is step *n*'s own action, the record answers "what goal does step *n*'s action serve?" and its first remaining step is the action that step performs.

Input	Source	Window
<code>next_actions</code>	summaries from step n onward; <i>the goal is inferred from these</i>	$n \rightarrow \min(n+19, N)$
<code>previous_entities</code>	extracted entities, accumulated and deduplicated per key	$\max(1, n-20) \rightarrow n-1$
<code>previous_goals</code>	goal names from this stage's own earlier output, consecutive duplicates collapsed	same
<code>recent_goal_with_steps</code>	the newest {goal, remaining steps} from this stage's own output	same
<code>previous_actions</code>	summaries, oldest first	$\max(1, n-10) \rightarrow n-1$

Table 2. Inputs to one goal record, with windows in case space.

Because `previous_actions` ends at $n - 1$ and `next_actions` begins at n , the two form one gapless sequence with the current step as the boundary. The prompt and the guide are concatenated into a single static system prefix so that prefix caching reuses the KV cache across requests; the per-step user message carries only the five fields above, around 500 tokens.



6.1 The goal vocabulary and the novelty escape hatch

The prompt carries a controlled vocabulary of canonical goal names, spanning intake and comprehension, billability, pricing, ERP charge entry, invoicing, filing and closure, plus operational goals, each with a definition and explicit *not* boundaries, followed by seven ordered disambiguation rules for actions that fit more than one goal.

The vocabulary was derived from the guide and **not validated against labelled data**. The prompt therefore instructs the model that when no canonical goal fits it must set a novelty flag and coin a name rather than force a bad fit, so a gap appears in the data instead of hiding

in a catch-all. In the reference run the novelty path fired on a small number of records under a single coined name, and all but two of the canonical goals appeared at least once.

6.2 Sequential within a case, and what that costs

Unlike every other generation stage this one is *serial within a case*: step n reads step $n - 1$'s output for goal continuity. Parallelism is across cases under one global limit, with cases launched longest-first so the longest chain is not queued behind short ones.

The run is therefore *tail-bound rather than throughput-bound*, and the arithmetic caps the value of extra capacity. The longest case is serial end to end, so no run completes in fewer sequential calls than that case has steps. Longest-processing-time scheduling reaches that floor at a modest number of concurrent slots; the default leaves slack for latency variance, and raising it much further leaves the makespan unchanged with most slots idle once only the long cases remain.

The same dependency makes this stage **resumable but not bit-reproducible**: the continuity fields are read from output files on disk, so a resumed run sees different context at the same step than a from-scratch run. Reproducibility requires deleting the output and rebuilding.

— 07

Stage: step-based question answering

This stage generates three question-answer pairs per step, teaching explanation, orientation and interface literacy.

Type	Teaches	Rotating axis
why_reasoning	why this is happening now: business rules, prerequisites, consequences of skipping	6 categories
workflow_navigation	position in the flow and what remains	macro / micro
visual_grounding	the interface: screen identity, control location, click path, on-screen state, look-alike controls, blockers	6 categories

Table 3. The three step-based question types.

A `visual_grounding` pair must be answerable *from* the current screen and *unanswerable without* one; a question that would read the same on any screen belongs to another type. Its six categories are screen identification (which application, screen and record, and what confirms it), control location (which tab, panel, row or dialog holds it), click path (the ordered route from here to the screen needed next), state reading (populated versus empty fields, active tab, selected row, blocking dialog, empty queue), affordance disambiguation (two near-identical controls and the visible cue that decides), and blocker recognition (a modal, a font-pack prompt, a session timeout, a truncated attachment list, a column behind a horizontal scroll, a scan needing rotation). The prompt requires control labels exactly as rendered, positions described relative to other on-screen elements rather than in pixels, and arrow-separated click paths wherever the guide defines one.

V0 Screen identification	V1 Control location	V2 Click path
V3 State reading	V4 Affordance disambiguation	V5 Blocker recognition

The step's own action summary is **deliberately withheld**, the opposite of Section 5. The product here is a question the trained model must answer from history plus the current screen (exactly its context at inference), so passing step n 's summary would let the generator write about an action the model cannot see. Chain-of-thought generation withholds nothing because there the revealed action *becomes* the target.

7.1 Category rotation as a Latin square

A prompt instructing the model to "cycle evenly through all six categories" cannot work in a stateless per-step call: each step independently picks whichever category its context suggests and the distribution collapses onto two or three. The assignment is instead derived from the step index and passed in as a block the prompt is ordered to obey. With $i = n - 1$:

$$\text{reasoning} = R_{i \bmod 6}, \quad \text{navigation} = \{\text{macro}, \text{micro}\}_{i \bmod 2}, \quad \text{visual} = V_{(i + \lfloor i/6 \rfloor) \bmod 6}.$$

The visual index is staggered by the block number so it does not move in lockstep with the reasoning index. This gives exact one-sixth coverage, needs no cross-step state, and is stable across re-runs. Measured over the full corpus, the reasoning categories are balanced to within a few percent of each other, the visual categories likewise, and **all 36 (reasoning, visual) pairings appeared**. Without the stagger only the six diagonal pairings would occur.

7.2 Degradation and repair

Missing screens: screenshot coverage measured over a spread sample is approximately 99.9%. A step without a readable screenshot still generates: the request carries no current-screen block, `visual_grounding` is removed from the required types, and the record holds two pairs instead of three, with its assignment block recording which types were requested.

Mislabelled types: because the prompt fixes the emission order, a pair whose type is missing or wrong is repaired by position rather than discarded; a pair beyond the required order (a visual-grounding pair on a step with no screen, say) has no position to map to and is dropped. Every repair and drop is logged.

`visual_grounding` is emitted last and is therefore the first casualty of a response truncated by the token cap, which makes truncation monitoring on a pilot a prerequisite to scaling up. Internal reasoning is enabled for this stage, unlike the goal stage, whose output is a short structured record rather than two long prose answers.

7.3 Prompt development, measured

The prompt was developed over eight pilot rounds. Each round generated 36 records: six cases × a six-step band, positioned at a different stage of the process, since six consecutive steps exercise every rotation axis once. Rounds one to seven used one set of six cases; **round eight used a disjoint set**, to distinguish a prompt that generalises from one fitted to the first six. Every round was scored with mechanical checks (banned phrasings, length percentiles, question anchoring, duplicate openers) and sample answers were read against their screenshots.

Round	Stage numbers	Unfilled placeholders	Assignment leak	Anchor as opener	Answer words (p50)	Defect rate	
1	41	19	3	7	89	64.8%	
2	0	3	16	14	171	30.6%	

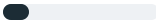
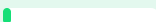
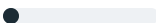
Round	Stage numbers	Unfilled placeholders	Assignment leak	Anchor as opener	Answer words (p50)	Defect rate	
3	0	2	8	1	167	10.5%	
5	1	5	0	0	160	5.6%	
7	0	3	0	0	165	2.8%	
8 (unseen cases)	1	2	0	4	165	6.5%	

Table 4. Pilot rounds for the step-based QA prompt.

In every case the fix was **removing the vocabulary the model was copying, not adding a prohibition**:

1. *Stage numbers*. Round one produced 41 references to numbered stages; a prose rule cannot outweigh 440 lines of numbered headings. An explicit number-to-name translation table covering all 23 stages took this to zero in round two.
2. *Assignment vocabulary echoed into answers*. Phrases such as "at this micro-level within..." appeared in 9 to 17 pairs per round and mean nothing to a reader who never saw the assignment. Tightening the prohibition did nothing; renaming the field itself (from a bare level label to a descriptive scope sentence) removed the words from the input and the echo stopped.
3. *Thin answers*. Answers ran to 89 words where the prompt demanded operational depth, because internal reasoning was consuming 4.6k of 5k completion tokens. Explicit per-type word budgets raised the median to roughly 165 words, where it has held.

Two fixes over-corrected and were relaxed: banning quadrant language outright (coarse position is useful when tied to a named container), and requiring the process anchor without saying where (it migrated into questions, making every opener identical). The residual ~3% is a handful of assignment-word slips and half-filled naming conventions with no new failure class. **Visual grounding needed no fixing**: every answer checked against its screenshot matched what was rendered, including window titles, filter breadcrumbs, system identifiers, column headers, tab strips and button groups, with no invented controls in any round.

Stage: process-knowledge question answering

The three stages above teach the process *as observed*. This one teaches it *as written*: its source is the refined guide, and it reads no case files, no summaries and no screenshots. It exists because observation cannot surface a gate the recorded window never triggered, a variant flow no recorded case took, an error path no operator hit, or a trap in the guide itself: a queue whose name contradicts its function, an absence that is correct by design. The two families are complements.

AS OBSERVED

Next-action, goal-based and step-based stages: case files, summaries, screenshots

AS WRITTEN

Process-knowledge stage: the refined guide only

8.1 Hand authoring, and why

The question bank is **written by hand**, questions *and* answers, against the guide: a deliberate rejection of generation for this one stage. Question generation is where a model quietly duplicates itself and leaves whole sections uncovered: it will write four questions about the most prominent decision fork and none about a conditional note on a rarely-taken branch. Answer generation has the matching failure: it paraphrases the paragraph nearest the question and misses the trap the question was built around.

Hand authoring buys two properties a sampling generator cannot.

Exhaustive

Every section of the guide is claimed by at least one question (the spine stages and substages, all five variant flows, all ten decision gates, the pricing-authority precedence, the evidence rules, the document mechanics, invoicing and consolidation, filing and dispatch, the multi-stage operator note, all fourteen error-handling entries and five blockers, the ten-row identifier map, the appendix and the nine cross-cutting notes).

Exclusive

No two questions ask the same thing, so no training target is duplicated under two identifiers.

Both are enforced rather than asserted: the loader aborts the build on a duplicate identifier, a duplicate question or a missing field, and a validate-only mode reports any entry a quality gate would drop, so an authoring slip surfaces before training rather than as a silent drop during the build.

Area	Area
Intake and case opening	Closure and operator notes
Gates and verdict	Variants and end-to-end flow
Non-billable close branch	Errors and blockers
Pricing	Identifiers and vocabulary
Evidence	Cross-cutting notes
ERP charge entry	Invoice
Filing and dispatch	

Table 5. Regions of the guide the question bank is organised into, one area file each.

The bank spans every stage of the guide and nineteen question types, with answers averaging around ninety-five words. Because nothing here is model-generated, building this stage's training data is a pure local transform: no API call, no network storage read, no endpoint. Its exhaustiveness is a claim about the revision it was written against: if the guide gains a section, the bank must gain an area file for it.

Two generators remain for additional paraphrase volume around the bank: one call per question type, and batches across seven subject categories. They are *optional*, require the inference endpoint, and carry none of the coverage guarantees above. Each sends a byte-identical system prefix of roughly 12.5k tokens with only the user message varying, so prefix caching serves it from cache after the first call; both stream, both resume by skipping units already on disk, and both salvage a truncated response back to its last complete object while recording that truncation occurred.

From records to training examples

Each generation stage produces *records* carrying the response, its provenance, the exact request that produced it and token accounting. A separate conversion stage per task turns records into *training examples*. The separation is deliberate: conversion is local, needs no GPU and no API, and finishes in minutes, so the message format, windows and system prompts can be revised without re-running inference.

9.1 A single uniform format

Every example, from every task, is a bare three-message JSON array and nothing else:

```
[
  {"role": "system",    "content": [{"type": "text", "text": "..."}]},
  {"role": "user",      "content": [{"type": "text", "text": "..."},
                                    {"type": "image_url", "image_url":
                                      {"url": "data:image/jpeg;base64, ..." } },
                                    {"type": "text", "text": "..."}]},
  {"role": "assistant", "content": [{"type": "text", "text": "..."}]}
]
```

There is no metadata wrapper. Provenance (case, step, origin tuple, context sizes and any stratification keys) lives in a parallel index file with one row per example in the same order, so a trainer never filters non-message keys out of an example and an analyst can still stratify an evaluation by any recorded property. Images are embedded as base64 data URIs rather than referenced by path, because the training data loader decodes only that form.

Task	Input	Output
Next-action prediction	previous actions with entities, current screen	<think> reasoning </think> , then the action
Next-goal prediction	case history, goals so far, goal in flight	{goal, remaining steps} as JSON
Situated question answering	case history, current screen, one question	the answer, as prose

Task	Input	Output
Process-knowledge QA	one question about the process	the answer, as prose

Table 6. The four training tasks derived from the generation stages.

The situated-QA conversion emits *one example per question-answer pair*, not one per step, because that is the task at inference: answer the question in front of you, not invent three of them. The shared screenshot is encoded once per step and reused across that step's examples. The process-knowledge conversion follows the same one-question-one-example contract, which keeps an exact correspondence between the audited bank and the training set, a property the optional trigger-phrase variants would break, and the reason they are off by default.

9.2 Context windows are inherited

Images are re-encoded with the *identical* bounds used at generation time (long side ≤ 1600 , short side ≤ 640 , JPEG quality 80), so the training image is the one the label was written against. A step with no readable screenshot yields an example with no current-screen block, matching what generation produced for it; the handful of steps affected contribute two situated-QA examples each instead of three, and their next-action examples train history-only reasoning.

Task	Lookback	Note
Next-action prediction	20	Matches the generator's own window; the reasoning was generated over exactly this history.
Next-goal prediction	10	Previous goals are drawn from the whole case before n ; the in-flight goal is step $n-1$.
Situated question answering	20	Not the goal task's 10: the answers were written against a 20-step window, so a shorter one would leave labels citing actions absent from their own input.
Process-knowledge QA	none	No case history, no screen, no retrieved document.

Table 7. Training-time context windows, in case-step space.

9.3 The leakage gates

Every system prompt is assembled at build time and asserted against a list of forbidden tokens; a match aborts the build. The lists are task-specific because the surplus context is.

The removals go further than deleting the guide. The next-goal prompt omits the canonical goal names, so the model learns its label space from labels rather than a listing; it also omits any process description, stage walkthrough or artefact-position table, leaving the workflow name as the only process term in the prompt, and its worked formatting examples use generic surfaces so a formatting lesson does not double as process instruction. The situated-QA prompt omits the question-type listing on the same reasoning (the question is self-describing) and carries the stage-name list only so the model has names to use instead of numbers. The next-action prompt omits every trace of the generation-time no-leak vocabulary, because a model with no hidden label has nothing to hide from.

Task	Removed	Gate
Next-action	the guide; all no-leak machinery	Workflow Guide , ground truth , next_action_text , evidence_steps , confidence score , no-leak , reasoning_prompt , target action , hidden label
Next-goal	the guide; the lookahead; the goal vocabulary	next_action , Workflow Guide , recent_goal_with_steps , lookahead
Situated QA	the guide; the category assignment block	Workflow Guide , next_action , lookahead , assignment , required_question_types , reasoning_category , navigation_scope , visual_category
Process-knowledge QA	the guide entirely	Playbook , playbook , Question Type , Canonical stage names , { { ,plus an eight-word shingle overlap check against the full guide

Table 8. What each conversion removes from the training prompt, and what its gate checks.

These gates are not decorative. Two documented instances:

The first draft of the situated-QA system prompt contained the rule "never cite a source: no 'according to the workflow guide'". The gate rejected it: forbidding the citation still told the model such a document existed. The rule was reworded; the gate was not loosened.

The next-goal gate was validated adversarially by injecting "consult the Workflow Guide" into the prompt template. The build raised rather than shipping.

The shingle check is the strongest of the four, because a token list can be satisfied by paraphrasing a leak and an eight-word run cannot. On a clean build it logs a 826-character system prompt sharing no eight-word run with the 42,675-character guide.

Because the system prompt is embedded in every example of its task, **editing a prompt makes that task's whole corpus stale** and requires a full rebuild. Two conversions therefore have no skip-existing mode: their index files must stay in exact row order with the examples, and a partial rebuild would desynchronise them.

— 10

Corpus assembly and splitting

The four task corpora are assembled into one flat directory with an 80/10/10 train/validation/test split, files renamed with a task prefix and case identifier so all four can share a directory without collision.

ONE CASE-TO-SPLIT ASSIGNMENT, APPLIED TO ALL CASE-DERIVED TASKS

Train 80%

Val 10%

Test 10%

Next-action, next-goal and situated QA: split by case, together

Process-knowledge QA: train-only, not case-derived

Two independent leakage risks force splitting by case rather than by example, and the second is easy to miss. **Within a task**, adjacent steps of a case share most of their context window, so a per-example split would put near-identical examples in train and test. **Across tasks**, all three case-derived tasks describe the same steps of the same cases, so splitting them independently would put a case in one task's train split and another task's test split: the same leak, invisible to any per-task audit. A single case-to-split assignment is therefore computed once and applied to all case-derived tasks.

Because case sizes span more than an order of magnitude, assignment is greedy: cases are sorted largest-first and each goes to whichever split is furthest below its example-count target. This keeps every case whole while landing the splits within a fraction of a percent of the nominal ratios, and sorting by size then identifier makes it deterministic without a seed.

The process-knowledge corpus is **train-only**. It is not case-derived (there is no case to hold out) and it teaches the process itself rather than any case, so its files keep their own authored identifiers rather than being renamed under a case.

A rebalancing mode changes the ratio of an existing build by moving *whole cases* between splits, only in the direction the new targets require and skipping any case large enough to overshoot badly, rather than re-copying the corpus. A second, narrower assembly for preference optimisation contains only the goal task and the process-knowledge QA, with most cases in train and a small number held out, and those held-out cases are *drawn from the first corpus's own validation split*, so a case held out for supervised fine-tuning is held out there too.

11

Verification

Verification runs over the full corpus wherever the check is cheap enough, and per sample otherwise. The following are the reference build's results.

Check	Result
Examples versus source goal records	all present
Coverage of every case step 1 ... N	none missing, none extra
Line-oriented corpus lines versus file count	match
System prompt size identical across examples	identical in all
Label equals the source record	all
In-flight goal equals step $n-1$'s record	all
Previous goals drawn from this case, steps $< n$	all

Check	Result
Forbidden tokens in the system prompt	none
Action list reconstructed from the case file (sampled)	all exact
Context quoting a step $\geq n$	none

Table 9. Next-goal task: verification over every example in the corpus.

The future-leakage check needs expanding, because a naive version misleads. An early sweep flagged a large share of a sample as quoting a step at or beyond the target. All were false positives: most were an identical summary genuinely recurring earlier in the window (operators repeat actions) and the rest were substring matches where a later summary is a prefix of an earlier, longer one. The window excludes the target step by construction.

Assembly verification checks four further properties: that the files on disk in each split are exactly the intended set, with missing and extra counts reported separately; that **no case appears in more than one split**; that a sampled file parses as a valid three-message array; and that the output root holds only the expected split directories. Any failure exits non-zero, so the build is usable as a pipeline gate.

Check	Result
Examples versus source records	all but one unparseable record
Coverage of every case step	only the pre-existing generation skips missing
Assistant text matches the reasoning-then-action shape	all sampled
Reasoning body equals the source trace exactly	all sampled
Action line equals the source summary exactly	all sampled
Index row origin equals the case file's own origin	all sampled
Index / line-corpus / per-file row alignment	all sampled, no mismatches
Bare three-message shape, no wrapper	all checked

Check	Result
Leak-guard terms in assembled examples	none in any sampled

Table 10. Next-action task: verification after the full build.

Each generation stage also records per-call token usage as an append-only log plus a recomputed summary, which makes a pilot run's per-call cost a basis for projecting a full run, the recommended check before committing to a corpus-scale run.

12

Limitations

Over half of the goal labels are copyable.

The in-flight goal shown to the model is step $n - 1$'s own record, which was generated with the lookahead, so for a continuation the answer is often already contained in it. Measured over the whole corpus, 35% of labels are exactly the previous record's remaining steps and a further 17% are a suffix of them: 52% copyable. This is the shrink-by-one mechanic the task is meant to teach, and it is not leakage of information the model lacks at inference, where that field holds its own previous prediction. But it does mean a large share of the signal is a copy-or-truncate task, and any evaluation on this split will be optimistic. A clean number needs a copy-free held-out subset, or a filter on the relation between the in-flight goal and the label.



■ Exactly the previous remaining steps, 35% ■ A suffix of them, 17%

The situated-QA labels inherit their generator's defects.

On the source corpus, 2.81% of pairs carry at least one: an unfilled placeholder (1.48%), a stage number (0.48%), a source citation (0.34%), a description of the image as an image (0.28%), or a generic control reference (0.20%). The conversion reformats and does not filter, so these pass through. The checks are cheap regular expressions over question and answer text, so filtering is available to anyone wanting a clean split.



Factual correctness against business reality is not verified.

Every generated label is grounded in the refined guide, so wherever the guide is wrong the training data is fluently and specifically wrong. A judged pass over a stratified sample (which the index files' category fields make easy to construct) is the check that has not been run.

The goal vocabulary was not validated against labelled data.

It was derived from the guide. The novelty flag and the per-goal frequency distribution are the instruments for finding its gaps, and both should be read before training: two canonical goals never fired, and a goal with no examples cannot be learned. The single coined name that did appear should be either promoted into the vocabulary or normalised away.

One stage is not bit-reproducible.

The goal generator reads its own prior output for continuity, so a resumed run sees different context at the same step than a from-scratch run.

Conclusion

The distance between a screen recording and a trainable corpus is largely a matter of deciding what the model must not be told. Generating good labels is comparatively easy once a written process description exists and a capable vision-language model can be handed a lookahead, a ground-truth action, or the guide itself. The difficulty is that each of those conveniences, left in the training prompt, converts a learning problem into a copying problem, and does so invisibly.

The methodology here treats that asymmetry as the central constraint. Seven stages (per-step semantic annotation, case boundary detection, guide refinement, next-action generation with synthesised reasoning, goal-based supervision from lookahead, screen-grounded situated question answering, and hand-authored process-knowledge question answering) run in dependency order, and the four that generate training tasks do so with surplus context and hand their records to a conversion that removes it and proves the removal with a gate that aborts the build. Splitting happens once at the case level and is shared across tasks, because the tasks describe the same steps. Every record resolves back to the screenshot and event it came from, every stage reports what it dropped and at what rate, and the properties that were not verified are named.

The resulting corpus (one uniform three-message format across every task) trains four complementary capabilities from a single body of observed work: what to do next and why, what the current activity is trying to achieve, how to read and operate the interface in front of it, and what the process says about situations no recording happened to contain. No stage depends on the particular process used here; each needs only recorded work that arrives as cases and a written description of the process to condition on.

NEXT-ACTION PREDICTION

What to do next and why

NEXT-GOAL PREDICTION

What the current activity is trying to achieve

SITUATED QUESTION ANSWERING

How to read and operate the interface in front of it

PROCESS-KNOWLEDGE QA

What the process says about situations no recording happened to contain

THE CENTRAL CONSTRAINT

The distance between a screen recording and a trainable corpus is largely a matter of deciding **what the model must not be told.**