

— ENTERPRISE WORKFLOW AUTOMATION

Agent Execution Mesh

A Plan-Execute-Arbitrate architecture for
autonomous enterprise workflow automation

AUTHOR

Neurologic AI

PUBLISHED

July 2026

CASE STUDY

Claim intake, Outlook and OpenKoda

ABSTRACT

Enterprise back-office work is defined less by any single system than by the seams between systems: a request arrives in a mailbox, is interpreted by a human, verified against a record in a line-of-business web application, entered into a form, annotated on a customer file, and answered by return mail. Conventional robotic process automation encodes such a sequence as a brittle script of coordinates and selectors, while a single large language model driving a browser directly must hold the entire procedure, the entire page, and the entire history in one context window, where it drifts, skips verification steps, and hallucinates. This paper describes the Agent Execution Mesh, a production system that decomposes

cross-application automation into three cooperating roles with strictly separated responsibilities: a deterministic coordinator that owns control flow and goal lifecycle, a language-model planner that reads a human-authored workflow guide and emits exactly one concrete goal at a time, and a language-model worker that executes that goal autonomously through a bounded loop of tool calls. The two language models run on separate endpoints and never share a context window; the coordinator, which decides what happens next, contains no model at all. Perception of the target web application is mediated by an ARIA accessibility snapshot with injected element references rather than by pixels or CSS selectors, so that every action is expressed in the semantic vocabulary of the page. We describe the state machine, the context-economy rules that govern what each agent is permitted to see, the tool layer, and the treatment of the workflow procedure as data rather than code. We then present a worked deployment (insurance claim intake spanning Microsoft Outlook and the OpenKoda web application) and report six real executions covering all four terminal branches of that workflow: 25 goals and 95 tool calls, of which 24 goals completed successfully, with the single goal-level failure correctly absorbed and converted into a customer response rather than an aborted run.

6

real executions,
all four branches

25

goals

95

tool calls

24

goals completed
successfully

01

Introduction

A large fraction of clerical work in insurance, healthcare administration, and financial services consists of moving a single case through several unrelated software systems. Consider a motor insurance claim. It arrives as free-form prose in an email. Somebody reads it, decides whether the sender supplied enough information to act on, looks up the referenced policy in a claims administration system, checks whether the policy was still in force on the day of the incident, opens a claim record if it was, writes a note on the

customer's file either way, and replies to the original message explaining what happened. No single application spans this procedure. The procedure exists only in the head of the clerk and, at best, in a training document.

Two families of automation address this and both have well-understood limits.

Classical robotic process automation records the clerk's keystrokes and replays them. It is fast and auditable, but it encodes the surface of the procedure rather than its logic. A relocated button, a renamed field, or an unanticipated branch (a policy number that does not exist, an email missing the claimant's name) breaks the script, because the script has no representation of intent from which to recover.

The opposite approach hands the whole problem to a single large language model equipped with browser and mail tools, and asks it to work through the case in one conversation. This is flexible but structurally fragile at production length. The model accumulates every page snapshot, every tool result, and every intermediate deliberation in one growing context. Verification steps that matter (comparing an incident date against a coverage end date, refusing to proceed when a required field is absent) are precisely the steps a model under context pressure elides, because they produce no visible progress. The model is simultaneously deciding what to do and how to do it, and the two decisions interfere: an agent deep in the mechanics of locating a dropdown is not the agent best placed to remember that the policy expired two months ago and the form should never have been opened.

The Agent Execution Mesh takes a third position. It observes that the two decisions above (what the next step should be, and how to carry that step out) have different inputs, different failure modes, and different context requirements, and that a third decision, when the procedure is over, should not involve a language model at all. The system therefore assigns each to a distinct component:

WHAT NEXT

The planner

The planner answers *what next*. It sees the workflow guide, the case context, and a compressed summary of everything already accomplished. It never sees a page snapshot, and it never touches a tool.

HOW

The worker

The worker answers *how*. It sees one goal, spelled out in concrete values, and the live state of the target application. It has no knowledge of the workflow guide and no authority to decide what comes after.

WHEN

The coordinator

The coordinator answers *when*. It is ordinary Python. It routes between the other two, promotes and archives goals, enforces the budget, and terminates the run.

We call the arrangement a *mesh* rather than a pipeline because control does not flow in one direction. Every transition passes back through the coordinator, so the topology is a star with a deterministic hub, not a chain. The planner and worker never address each other; they communicate only through state the coordinator has inspected and, where appropriate, edited.

Every transition passes back through the coordinator, so the topology is **a star with a deterministic hub**, not a chain.

The remainder of this paper describes the state machine and its three nodes (Section 2), the shared state object and the rules governing what each agent may see (Section 3), the tool layer and the accessibility-based perception model (Section 4), the treatment of the workflow procedure as data (Section 5), the deployed claim-handling case study (Section 6), observed executions (Section 7), observability (Section 8), and the design rationale, and what the deployment establishes together with the roadmap it suggests (Sections 9 to 10).

Architecture

2.1 The state machine

The system is implemented as a compiled state graph with three nodes and a single shared, typed state object. The coordinator is the entry point. Both the planner and the worker return unconditionally to the coordinator, and only the coordinator may reach the terminal state. Figure 1 shows the topology.

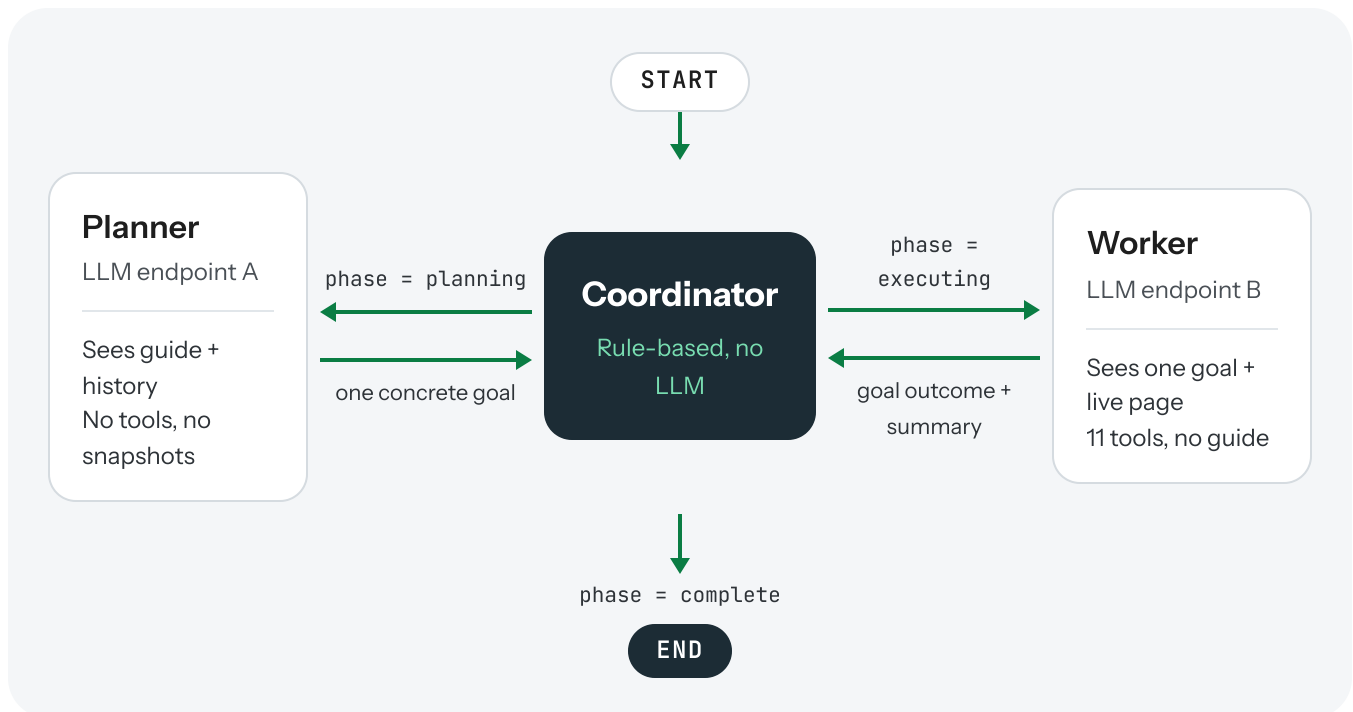


Figure 1. The mesh topology. Control always returns to the deterministic coordinator; the planner and worker never communicate directly and run on separate model endpoints.

A single field, `phase`, drives routing. It takes three values (`planning`, `executing`, and `complete`) and the router is a six-line function with no model call and no heuristics. The consequence is that the control flow of the system is fully inspectable: whatever the language models produce, they cannot change where control goes next, only what the coordinator is asked to decide about.

2.2 The goal as the unit of work

The atom of the system is the *goal*: a named, self-contained unit of progress that the planner emits and the worker consumes. A goal is a plain dictionary and carries its own execution record, so that a completed goal is simultaneously a plan, a result, and an audit

trail. Table 1 lists its fields and Table 2 its lifecycle.

Field	Contents
<code>name</code>	Short human-readable label, e.g. “Policy Verification in Openkoda”
<code>extra_details</code>	Fully concretised instructions: URLs, credentials, policy identifiers, field values, dates. This is the planner's entire channel to the worker.
<code>goal_state</code>	<code>IN_PROGRESS</code> , <code>ACCOMPLISHED</code> , or <code>FAILED</code>
<code>actions_taken</code>	Ordered list of {tool, arguments, response} records, including failed calls
<code>summary_of_work</code>	Natural-language report written by the worker; the only thing the planner will later read about this goal
<code>done</code>	Set by the planner instead of a goal to declare the workflow finished

Table 1. Structure of a goal.

Transition	Effected by	Trigger
$\emptyset \rightarrow$ proposed	Planner	Model emits a JSON goal object
proposed $\rightarrow$ <code>IN_PROGRESS</code>	Coordinator	Goal accepted; action list initialised to empty
<code>IN_PROGRESS</code> $\rightarrow$ <code>ACCOMPLISHED</code>	Worker	<code>goal_complete(is_accomplished=True)</code>
<code>IN_PROGRESS</code> $\rightarrow$ <code>FAILED</code>	Worker	<code>goal_complete(is_accomplished=False)</code> , model-endpoint error, or action budget exhausted
terminal $\rightarrow$ archived	Coordinator	Goal appended to <code>completed_goals</code> ; counter incremented

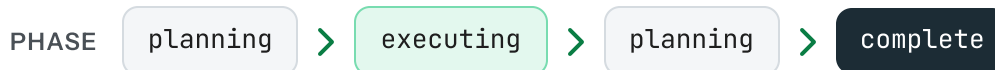
Table 2. Goal lifecycle. Note that only the coordinator performs promotion and archival; the worker may only report an outcome.

2.3 Coordinator

The coordinator is deliberately the least intelligent and most important component. It is roughly 150 lines of Python containing no model call. On each invocation it examines `phase` and the current goal and does one of five things.

If the phase is `planning` and no goal is present, it is the first iteration; it routes to the planner. If the phase is `planning` and the planner has returned a goal, the coordinator promotes it (stamping `IN_PROGRESS`, clearing the action list, opening a log section) and switches the phase to `executing`. If the phase is `planning` and the planner returned `done: true`, the run ends. If the phase is `executing` and the current goal has reached a terminal state, the coordinator archives it, performs the selective context enrichment described in Section 3.3, increments the goal counter, checks it against the budget, and returns the phase to `planning`. Finally, if the goal budget is exhausted, the run ends regardless of what the planner intended.

Resolution (whether the case as a whole succeeded) is likewise computed by the coordinator rather than asserted by a model. It is defined as the conjunction of the states of all archived goals: a run is resolved if and only if every goal reached `ACCOMPLISHED`. Section 10 discusses where this definition is too strict.



2.4 Planner

The planner is invoked once per goal. It is given the workflow guide verbatim, the current case context, and a rendered summary of every completed goal in the form `[STATE] name` followed by that goal's `summary_of_work`. It is asked to emit a single JSON object and nothing else.

Three properties of this design carry most of its weight.

1

One goal at a time.

The planner is never asked for a plan, only for the next step. A full plan produced up front would be a prediction about a system state the planner has not yet observed, and every subsequent observation would require replanning. By emitting one goal and then yielding, the planner replans implicitly at every step, at the cost of one model call per goal. Conditional branches (expired policies, missing fields, unknown policy numbers) therefore need no special machinery: they are simply what the planner decides after reading the relevant goal summary.

2

Concretisation is the planner's job.

The worker cannot see the workflow guide. Anything the worker needs (a URL, a password, a policy identifier, a date to type into a form) must appear literally in `extra_details`. The prompt makes this explicit and reinforces it with paired correct/incorrect examples, because a vague goal such as “log into the system using the credentials from the case context” is unexecutable by a worker that has been shown only a 1500-character context window. This is enforced by construction rather than by convention: the guide is simply absent from the worker's prompt.

3

No fabrication.

The planner is instructed that values may be copied only from completed goal summaries or the case context, and that a field reported as “not found” must be propagated as “not found” rather than replaced by a plausible substitute. The prompt carries explicit negative examples of this failure, because the temptation to fill a table cell is strong and the resulting error (telling a customer that a detail they supplied was missing) is customer-visible.

The planner is called through a direct HTTP request to an OpenAI-compatible chat completions endpoint at temperature zero, and its reply is parsed with a small extractor tolerant of fenced code blocks. A parse failure is not fatal: it is converted into a synthetic `done` goal carrying the error text, so the graph terminates cleanly and the report records why.

2.5 Worker

The worker receives one goal and executes it in an internal loop bounded at 25 actions. Unlike the planner it uses native tool calling: the eleven task tools plus `goal_complete` are bound to the model, and the model's structured tool-call output is dispatched directly. Its loop is the following.

- 1 Invoke the model on the running conversation.
- 2 If the reply contains no tool calls, treat the text as a completion report and exit.
- 3 Otherwise, for each tool call: if it is `goal_complete` , record the outcome and exit; if it is any other tool, execute it, normalise the result to a string, append it to `actions_taken` , and append a tool message to the conversation so the model observes the result.
- 4 Repeat.

Two details are worth drawing out.

The conversation is real, not reconstructed.

A common shortcut in agent implementations is to rebuild the prompt on every iteration from a summary of past actions. This system instead appends to a genuine message list, so the model sees its own prior tool calls and their verbatim results. Failures therefore remain visible in-context, which is what allows the worker to self-correct within a goal: attempting `fill` on an element that turns out to be a select control, seeing the error, and retrying with `select_option` . Section 7 shows this behaviour in a production run.

Completion is an explicit tool call.

The worker signals that it is finished by calling `goal_complete(summary, is_accomplished)` . Making completion a tool rather than an out-of-band text convention keeps the model in tool-calling mode for the entire goal, which measurably reduces the failure in which a model narrates an action (“I will now click the Login button”) instead of emitting a call. A bare text reply is still accepted as a fallback, but it is the degraded path.

The summary the worker writes at completion is the only thing that survives into the planner's view of the goal. It is therefore not a courtesy but a structural interface, and the worker prompt specifies its contents: identifiers, extracted values, dates read from detail pages, and (for email goals) the message identifier, without which no later goal can reply in the same thread.

State and context economy

3.1 The shared state

All three nodes read and write a single typed dictionary, summarised in Table 3. Only `messages` accumulates by concatenation; every other field is overwritten by the node that owns it. This keeps the routing surface small and makes it obvious which component is responsible for which field. The state is not small in bytes (because each archived action stores the verbatim tool response, including full page snapshots, the goal archive of a seven-goal run serialises to roughly 130 KB), but the fields that drive control flow are a handful of scalars, and no agent is shown the archive in raw form.

Field	Written by	Purpose
<code>case_context</code>	main, coordinator	Case-specific facts: target URLs, credentials, per-case instructions, and email data promoted by the coordinator
<code>workflow_guide</code>	main	The procedure document; read only by the planner
<code>completed_goals</code>	coordinator	Archive of terminal goals with full action records
<code>current_goal</code>	planner, coordinator, worker	The goal in flight
<code>phase</code>	all	Routing discriminator
<code>goal_count</code>	coordinator	Budget counter
<code>is_resolved</code>	coordinator	Terminal verdict
<code>messages</code>	all	Append-only narration for the log

Table 3. The shared state object.

3.2 Asymmetric visibility

The most consequential design decisions in the system are about what each agent is *not* shown. Table 4 states the rules.

	Planner	Worker	Coordinator
Workflow guide	full	never	never
Case context	full	first 1500 characters	full
Completed goals	names, states, summaries	names, states, summaries	full records
Page snapshots	never	live, every action	never
Raw tool results	never	full, in conversation	never
Tools	none	11 + <code>goal_complete</code>	none

Table 4. What each agent can see. The blanks are load-bearing.

The planner is denied page snapshots because an ARIA snapshot of a moderately complex form page runs to thousands of tokens of markup that is irrelevant to the question “what should happen next”. Admitting it would displace the workflow guide and the goal history: exactly the material on which planning quality depends. The planner reasons over *summaries of outcomes*, which is the correct abstraction level for procedural decisions: “Coverage End Date: 2026-06-21” is the fact that matters, not the accessibility tree of the page that displayed it.

Symmetrically, the worker is denied the workflow guide. Given the guide, a capable model will read ahead and continue past its instructions: logging in when it was asked only to read an email, filling a form when it was asked only to verify a policy. Withholding the guide makes overreach mechanically harder, and the worker prompt reinforces it: complete the stated actions, then stop, because a planning agent decides what comes next.

3.3 Selective context enrichment

Some information genuinely must flow forward as input to later goals rather than as a summary of past ones. When a claim email is parsed, the extracted claimant name, policy number, incident date, contact address, and message identifier are not merely a record of work done; they are the raw material every subsequent goal consumes.

The coordinator handles this with a narrow, explicitly scoped rule. When it archives an accomplished goal, it inspects the tools that goal actually used. If any belonged to the email tool family, the goal's summary is appended to `case_context` under a labelled heading;

otherwise the summary reaches the planner only through the ordinary goal history. The predicate is over observed tool usage, not over the goal's name or the model's own judgement, so it cannot be talked into firing.

EVERY GOAL SUMMARY
Unbounded context growth; stale observations compete with authoritative ones

AS IMPLEMENTED
Promote only goals that used email tools

NEVER ENRICH
Every goal re-derives the case facts from the history

This is a deliberate compromise between two failure modes. Enriching context with every goal summary produces unbounded context growth and, worse, lets stale intermediate observations compete with authoritative ones. Never enriching forces every goal to re-derive the case facts from the history. The rule as implemented promotes exactly the class of data that is extracted from outside the system and needed verbatim downstream.

3.4 Termination

Four independent mechanisms bound a run, listed in Table 5. This layering is intentional. The planner's `done` flag is the semantic termination condition and the one that fires in healthy runs; the remaining three are structural backstops that hold even if the planner is looping, degenerate, or unavailable.

Bound	Value	Effect on exhaustion
Planner <code>done</code> flag		Normal termination; resolution computed from goal states
Goals per run	10	Run closed; resolved only if every goal accomplished
Worker iterations per goal	25	Goal marked <code>FAILED</code> ; planner may re-plan once
Graph recursion limit	200	Hard stop enforced by the graph runtime

Table 5. Execution bounds, as configured in the deployed system. The worker bound counts model invocations rather than recorded actions; a single iteration may dispatch more than one tool call.

The tool layer

Eleven tools are exposed to the worker, split across two very different modalities: a browser driven through the accessibility tree, and a mail system driven through a REST API. A thin dispatcher maps tool names to implementations and bridges the synchronous agent loop to the asynchronous tool implementations, so that adding a tool requires touching one map and one declaration, and no agent code.

BROWSER, VIA THE ACCESSIBILITY TREE

snapshot navigate fill click
select_option press_key

MAIL, VIA A REST API

fetch_unread_emails get_email
search_emails mark_as_read
send_email

4.1 Perception through the accessibility tree

The browser tools do not use CSS selectors, XPath expressions, or screen coordinates. Instead, every browser tool returns an ARIA accessibility snapshot of the current page: the same semantic tree a screen reader consumes, rendered as indented YAML in which each node carries a role and an accessible name.

Before the snapshot is handed to the model, a reference-injection pass walks it and assigns a short identifier (`e0` , `e1` , `e2`) to every node whose role is interactive: buttons, links, textboxes, comboboxes, checkboxes, options, tabs, and similar. Each identifier is recorded in a table mapping it back to the triple (role, accessible name, ordinal), where the ordinal disambiguates repeated role-name pairs such as several “Edit” buttons in a table. Interaction tools take an identifier and resolve it back to a live element by role and name.

01

ARIA snapshot

Role and accessible name per node



02

Reference injection

`e0` , `e1` , `e2` on interactive nodes



03

Reference table

(role, accessible name, ordinal)



04

Resolve and act

Live element found by role and name

This indirection buys three things. It gives the model a compact handle to refer to: `ref="e24"` rather than a brittle selector string. It guarantees that a resolved element is the one the model actually saw, because the mapping is rebuilt on every snapshot. And because resolution goes through role and accessible name rather than through the DOM, the locators survive the class-name churn and markup restructuring that breaks conventional selectors: a Login button remains a button named “Login” across a front-end refactor.

The cost is that identifiers are valid only until the next snapshot. The system accepts this and makes it harmless by *fusing action and observation*: every browser tool returns a fresh, re-referenced snapshot as part of its own result. The model never needs a separate observation step after acting, the identifiers it holds are never more than one action old, and the prompt can enforce a simple discipline: snapshot first, then act, and never snapshot twice in a row. The one gap is the error path, which returns an empty snapshot alongside the error text; after a failed action the model is therefore holding identifiers from before that action, and must snapshot again if the failure may have changed the page.

4.2 Uniform results and observable failure

Every tool returns a dictionary with a status field, a human-readable message, and, for browser tools, a snapshot. The worker normalises this into the string the model sees. Successful calls yield the message plus any payload; failed calls yield a string beginning `Error:` followed by the underlying exception text.

Failures are not swallowed and not retried by the framework. They are recorded in `actions_taken` exactly like successes and inserted into the conversation so the model reads them. This is the mechanism by which self-correction works at all: the framework's contribution is to make failure legible, and the model's contribution is to respond to it. A framework that retried silently would deny the model the information it needs to change strategy.

Tool	Arguments	Behaviour
BROWSER		
snapshot	(none)	Returns the referenced accessibility tree of the current page
navigate	url	Loads a URL, waits for network idle, returns the new snapshot

Tool	Arguments	Behaviour
<code>fill</code>	element, ref, text	Types into a resolved input; returns the resulting snapshot
<code>click</code>	element, ref	Clicks a resolved element, waits for load, returns the snapshot
<code>select_option</code>	element, ref, value	Chooses an option in a native select control
<code>press_key</code>	key	Sends a key event; used to submit search fields
MAIL		
<code>fetch_unread_emails</code>	max_count, subject_filter	Lists unread messages with identifiers and previews
<code>get_email</code>	message_id	Retrieves the full body of one message
<code>search_emails</code>	query, max_count	Keyword search across subject and body
<code>mark_as_read</code>	message_id	Clears the unread flag
<code>send_email</code>	to, subject, body, reply_to_message_id	Sends a message, threaded when a parent identifier is supplied

Table 6. The worker's tool surface. The `element` argument on interaction tools is a natural-language description used only for logging; resolution is by `ref`.

4.3 Mail as an API, not a browser

Mail operations run against the Microsoft Graph API through a delegated interactive sign-in with a serialised token cache, so that consent is given once and subsequent runs acquire tokens silently. This is a deliberate asymmetry. Reading a mailbox through a web client would consume tens of actions per message in navigation, scrolling, and rendering, and would subject the run to layout changes in a product the operator does not control. The API returns the same information in a single call with a stable schema.

The general principle the system follows is that a browser is a fallback interface, used when a system offers no programmatic one. The target line-of-business application in the case study below has no usable API, so it is driven through the accessibility tree; the mailbox has an excellent one, so it is not.

05

The workflow guide as data

The procedural knowledge of a deployment lives in a plain-prose document, loaded as a string and handed to the planner. It is not code, and it is written in the register of staff training material rather than of an API. Its sections describe the purpose and end state, the applications involved and how to navigate them, the data fields and their formats, the decision points, a numbered step-by-step procedure, a decision playbook, data-entry rules, edge cases, and recovery guidance.

GUIDE SECTIONS

Purpose and end state

Applications and navigation

Data fields and formats

Decision points

Step-by-step procedure

Decision playbook

Data-entry rules

Edge cases

Recovery guidance

Two consequences follow. First, the document is writable and reviewable by the operations staff who own the process, not only by engineers; it reads like the training material such a process already has. Second, retargeting the system to a different procedure is authoring work rather than software work: a new guide, a case context describing the new systems, and (only if genuinely new capabilities are required) new tools.

The prompts, by contrast, encode invariants that hold across all workflows. Roughly four hundred lines of prompt specification carry rules of five kinds:

01 Gateways

Preconditions that must hold before an expensive or irreversible phase begins. In the deployed workflow, three fields must be present in the inbound message before the system is permitted to touch the target application at all.

02 Distinctions

Pairs of situations that are superficially similar and must produce different outcomes. The archetype is *missing* versus *invalid*: a claim with no policy number and a claim whose policy number does not exist require different customer responses, and conflating them produces a reply that is both wrong and confusing.

03 Evidence requirements

Rules stating that a value may be reported only if it was actually read from a specific page, with list views explicitly disallowed as sources for detail fields.

04 Outcome consistency

Rules binding the content of an outward communication to what the system actually did: if a claim was created, the reply confirms it; if the form was never opened, the reply explains why.

05 Scope discipline

Rules keeping the worker inside its goal, and the planner to one goal per turn.

Most rules are stated with a positive and a matching negative example; the negative examples are drawn from failures observed in development, which makes the prompt file a cumulative record of the ways the system has previously been wrong. The rules added most recently tend to be the ones still stated in the positive only, which is a reasonable proxy for how well understood each failure mode is. It is, at 376 lines, the single largest module in the codebase: a fair reflection of where the engineering effort in a system of this kind actually goes.

06

Case study: claim intake

across Outlook and OpenKoda

6.1 The procedure

The deployed workflow processes inbound motor insurance claims. Claims arrive as free-form email. The target system of record is OpenKoda, a web application with Policy, Claim, and Client sections and no automation API. The procedure is summarised in Figure 2.

The system fetches unread messages matching a subject filter and processes exactly one (the oldest) per run, an isolation property that keeps a failure on one claim from contaminating another. It reads the body and extracts the claimant's name, policy number, incident date and time, description, contact address and phone, vehicle details, and claim amount, distinguishing throughout between the sender of the message and the claimant described in its body, which are frequently different parties.

The first gateway then applies. If the claimant's name, the policy number, or the claimant's contact address is absent, the system does not open the target application at all. It replies to the original sender with a table of every extracted field, showing the values it did find and marking only the genuinely absent ones, and closes the case.

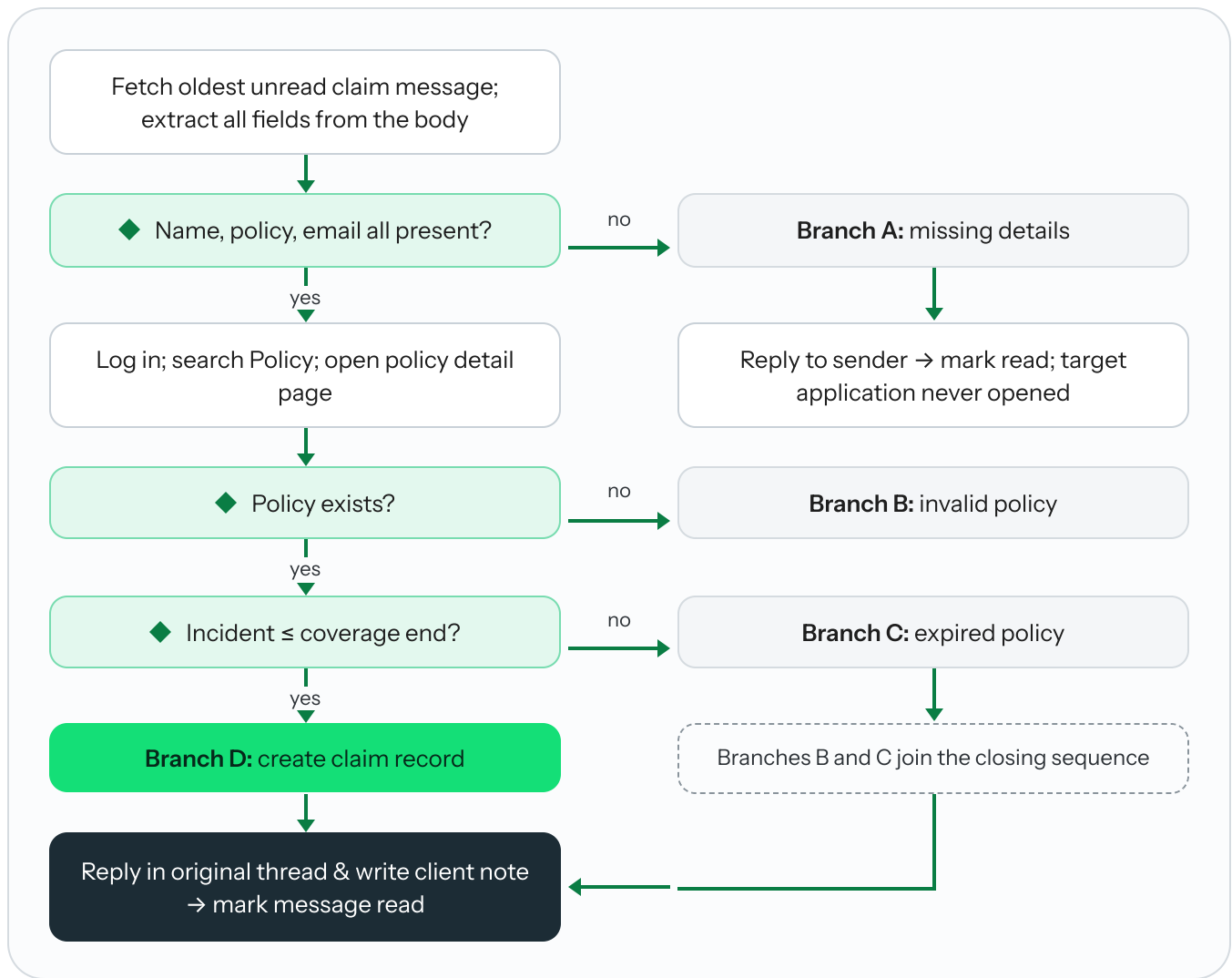


Figure 2. The four terminal branches of the deployed claim workflow. Branch A terminates before the target application is opened; the remaining three converge on a common closing sequence.

If the gateway passes, the system logs in, searches the Policy section for the exact policy identifier, opens the policy *detail page* (the list view is explicitly not accepted as a source) and reads the coverage end date. Two branches follow. If no such policy exists, the claim is rejected as an invalid policy number, with a reply that says so rather than claiming details were missing. If the policy exists but the incident date falls after the coverage end date, the claim is rejected as expired, quoting both dates.

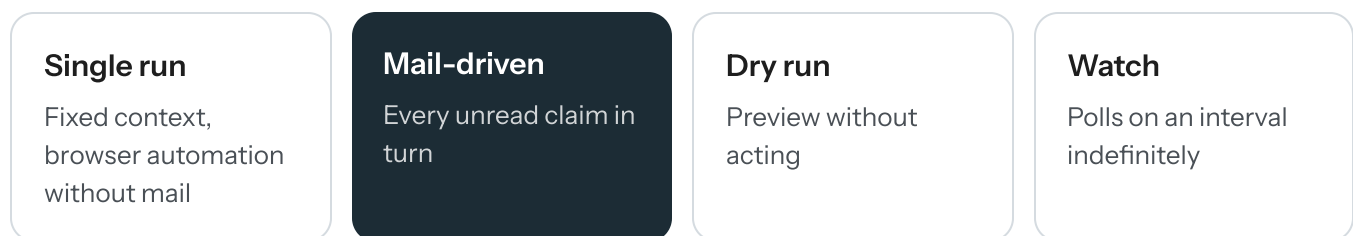
Only when the policy exists and was in force is the claim form opened and submitted with reported date, description, amount, currency, status, and policy identifier.

The three branches that reach the target application converge on a common closing sequence: reply in the original mail thread, locate the claimant in the Client section and write a note recording the disposition, and mark the message read. Branch A closes more cheaply,

because there is nothing to record against a customer the system never looked up: it replies and marks the message read without touching the application. The ordering of reply and note within the closing sequence is not fixed by the guide, and the planner chooses it per case; Section 7 shows the claim-created branch writing the note before the confirmation, and both rejection branches replying first.

6.2 Operating modes and session reuse

The entry point offers four modes: a single run against a fixed context for testing browser automation without mail; a mail-driven mode that processes every unread claim in turn; a dry run that previews what would be processed without acting; and a watch mode that polls on an interval indefinitely.



Because the browser process outlives an individual claim, a session-reuse optimisation applies. Once any claim in a session has logged in, later claims receive a note in their case context telling the planner to skip the login goal and confirm state with a snapshot instead. The flag is set from *observed* evidence (a completed goal whose name indicates a successful login) rather than from the loop index, precisely because Branch A terminates before any login occurs and would otherwise set the flag falsely.

This optimisation is also a place where a shortcut could quietly become a correctness bug, and the prompts treat it accordingly. The note is scoped in the strongest available terms: it permits skipping login and nothing else. The required-fields gateway and the expiry comparison still apply in full, and the prompt carries worked negative examples showing a model incorrectly using the already-logged-in note as licence to bypass them. Saving four actions is not worth processing an expired claim.

Observed executions

We report six consecutive runs from a single mail-driven cycle against a live Outlook mailbox and a running OpenKoda instance. All four terminal branches are represented, and the three gateway rejections happen to fail on three *different* required fields, which exercises the gateway more thoroughly than a repeated case would. Table 7 summarises the cycle.

Run	Branch	Goals	Acc.	Failed	Actions	Tool calls	Wall clock
1	D: claim created	7	7	0	45	38	240 s
2	B: invalid policy number	5	4	1	27	22	180 s
3	C: expired policy	5	5	0	29	24	176 s
4	A: no contact email	3	3	0	7	4	134 s
5	A: no claimant name	2	2	0	5	3	104 s
6	A: no policy number	3	3	0	7	4	110 s
Total		25	24	1	120	95	

Table 7. Six consecutive runs covering all four branches. *Actions* counts every entry recorded in `actions_taken`, including each goal's terminating `goal_complete` call; *tool calls* excludes those, and so counts only work done against Outlook or the browser. Wall clock is derived from consecutive report timestamps; the figure for run 6 is confirmed against the logged session duration of 110.45 s. Run 2 is the only unresolved run, for the reason discussed in Section 7.2.

7.1 Branch D: the full path

Run 1 processed a valid claim in seven goals: read the mail, log in, verify the policy, fill the claim form, write the client note, send a confirmation, mark read. The planner read a coverage end date of 2026-08-30 from the verification summary, compared it against an incident date of 2026-08-01, and proceeded. The claim was created with identifier 120508 and verified against the claims list before the goal was reported complete.

This run also exercises one of the subtler prompt invariants. The inbound email gave no claim amount (the extraction summary records “Claim amount: not found”) and the worker supplied a default of 1000 when filling the form, because the form requires the field. The

confirmation email is consequently required to quote the value that was submitted rather than the value that was extracted, and it does. Without that rule the customer would receive a confirmation whose amount field disagreed with the record just created on their behalf.

The action distribution across the seven goals is informative. Reading the mail took three actions and sending the confirmation two, because mail is driven through an API: two Graph requests to list and read, one to send, each followed by the completion call. Login took five. Policy verification took six, including the mandated click through to the detail page. Filling the claim form took twelve. Writing the client note took fifteen: three more than the claim form itself, and the most expensive step in the workflow. Locating a customer record and attaching a note to it involves more navigation than filing the claim it describes. That the accounting makes this visible at all is a direct product of recording every action against the goal that motivated it.



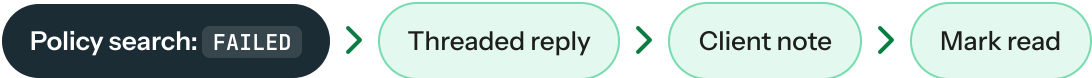
Figure 3. Recorded actions per goal in run 1, for the goals whose counts are reported above. Writing the client note is the most expensive step in the workflow.

7.2 Branch B: a failure that is not an error

Run 2 is the most instructive. The inbound message supplied policy number 120712-XYZ, which does not exist in the target system. The worker navigated to the Policy section, typed the identifier into the section's own search box, submitted it, observed an empty result set, and called `goal_complete` with `is_accomplished=False` and a summary saying exactly that: the policy could not be found and may not exist. Six recorded actions, no fabricated result, and no attempt to press on to a claim form that had nothing to attach to.

The planner then did the thing the architecture exists to make possible. It read a `FAILED` goal and, rather than retrying it or aborting the case, recognised the failure as diagnostic. A policy number that was supplied but does not exist is a different customer situation from a policy number that was never supplied, and it calls for a different reply. The planner

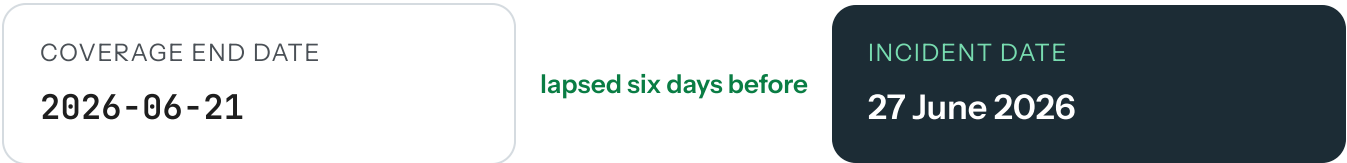
produced three further goals, all accomplished: a threaded reply asking the sender to verify and resubmit the number, a client note recording the rejection against Claire Bennett's record, and the mark-read. The distinction the prompts spend most of their length defending (missing versus invalid) is made here at exactly the moment it matters, and it is made by a component that never saw the page on which the evidence appeared.



The run is nonetheless recorded as unresolved, because resolution is defined as every goal having succeeded. This is the correct conservative default (a run containing a failure should not silently claim success) but it does mean the flag under-reports outcomes on precisely the branches where a failure is the expected signal. Section 10 describes the reporting change that resolves it.

7.3 Branch C: expiry, and cross-run session reuse

Run 3 handled a claim whose policy had lapsed six days before the incident. The planner read a coverage end date of 2026-06-21 from the detail page against an incident date of 27 June 2026, correctly declined to open the claim form, and planned a rejection quoting both dates. The client note it then wrote records both dates as well, so the rejection is auditable from inside the system of record and not only from the mailbox.



Runs 2 and 3 together demonstrate session reuse: neither contains a login goal. The browser authenticated once in run 1 and stayed authenticated for both subsequent claims; the entry point's evidence-based flag was set from run 1's completed login goal, and the planner skipped straight to policy verification in each case, while still performing the gateway check and, in run 3, the expiry comparison in full. Two logins saved across three application-touching claims, with no relaxation of either safety check.

7.4 Branch A: the gateway pays for itself, three different ways

Runs 4 to 6 were all rejected at the gateway, and each failed on a different one of the three required fields: run 4 had a claimant name and a policy number but no contact address, run 5 had no claimant name, and run 6 had no policy number. In each case the planner read

the extraction summary, applied the gateway, and planned a reply directly. No browser was opened.

The replies illustrate the no-fabrication rule concretely. Run 4's reply table shows `Contact Email: None` while preserving `Contact Phone: 703-273-5388`; the field that was genuinely absent is marked absent, and the fields that were present are copied verbatim rather than blanked along with it. This sounds trivial and is not: the failure mode it guards against, telling a customer that details they did supply were missing, is both customer-visible and the natural thing for a model to do when it is filling a template.

The cost asymmetry is the argument for putting the gateway where it is. A gateway-rejected claim costs three or four tool calls; the fully processed claim in run 1 costs thirty-eight. Deciding early that a case cannot be completed is worth roughly ten times its own cost, and it removes any possibility of a half-entered record in the system of record, a partially filled claim form being considerably worse than no claim form.

One inconsistency is visible across these three runs, and it is worth stating plainly because it is invisible in the runs' own verdicts. Runs 4 and 6 planned the mark-as-read goal that both the guide and the planner prompt call for; run 5 returned `done` immediately after the reply and left the message unread, where a later poll will pick it up again. All three runs report as fully resolved. Section 10 describes the report-driven regression check that would surface this automatically.

7.5 Aggregate observations

Across 95 tool calls the framework itself contributed no retries: every recovery visible in the logs was a model responding to an error string the framework had made legible. The worker's per-goal iteration budget of 25 was approached but never exhausted (the most expensive single goal recorded 15 actions) and the goal budget of 10 was never reached, though run 1's seven goals leave less headroom than is comfortable for a workflow that could plausibly grow another branch.

The wall-clock column carries a caution that the tool-call column alone would hide. Run 3 issued 24 tool calls in 176 seconds; run 5 issued 3 in 104 seconds. Eight times the work for less than twice the elapsed time, because elapsed time is dominated by model latency (one planner call per goal plus one worker invocation per action) rather than by the browser or the Graph API. The gateway therefore saves roughly nine-tenths of the *tool calls* on a rejected claim but under half of the *elapsed time*. Both figures are real, and they point at different

optimisations: fewer tool calls reduce the surface on which the automation can go wrong, while reducing latency requires fewer or faster model round trips, which is a question about goal granularity rather than about the workflow.

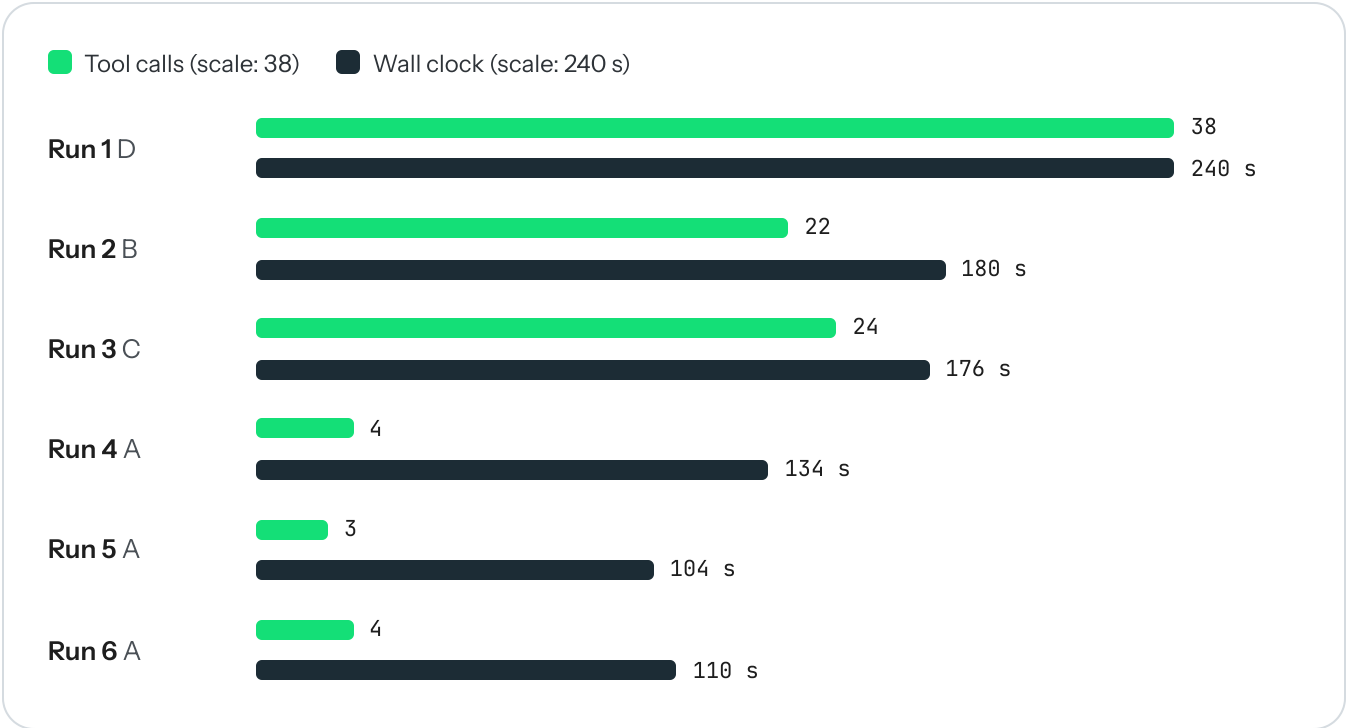


Figure 4. Tool calls and wall-clock time per run, from Table 7. Elapsed time is dominated by model latency rather than by the browser or the Graph API.

08

Observability

Every run produces two artifacts. The first is a chronological log recording each agent entry and exit with wall-clock duration, each goal boundary, each tool call with its arguments, each result with its status and latency, every error, and a session summary. This log is written to a single fixed path and is truncated at the start of each run, so it is the operator's live view of the run in progress. The second artifact is the durable one: a timestamped, structured JSON report containing the initial and final case context, the complete list of goals with every action and its verbatim response, and counts of goals created, accomplished, failed, and abandoned.

LIVE VIEW

Chronological log

Single fixed path, truncated at the start of each run

DURABLE

Structured JSON report

Timestamped; every goal, every action, every verbatim response

Because every action is recorded against the goal that motivated it, the report supports a form of analysis that flat agent transcripts do not: cost can be attributed to procedural steps rather than to the run as a whole. The observation in Section 7 that writing the client note is the most expensive operation in the workflow is a direct read of the report, and it is exactly the kind of finding that redirects optimisation effort: there is more to gain from a dedicated note-writing path than from shaving actions off login.

The report format is also the natural input to regression testing: a report is a complete, replayable description of what the system did, and two reports for the same branch are directly comparable.

09

Design rationale

01 Determinism where determinism is possible

The component that decides control flow contains no language model. Routing, promotion, archival, budget enforcement, and the resolution verdict are ordinary code. Only the two genuinely open-ended questions (what should happen next in this case, and how do I accomplish this on this page) are delegated to models. This is not a concession to model unreliability so much as a statement about which parts of the problem are actually ambiguous.

02 Separation of what from how

The planner and worker are not merely two prompts; they are two models on two endpoints with disjoint context, tuned for different tasks. The planner must produce a small, correct JSON object after reasoning over procedural text; the worker must emit long chains of accurate tool calls against live markup. Nothing requires these to be the same model, and the split allows each to be sized, replaced, or specialised independently. It also means neither pays the context cost of the other's material.

03 Plan lazily

One goal at a time is more model calls than a batch plan, and it is strictly better in a stochastic environment. Each goal is planned against observed reality rather than a prediction, and conditional branching requires no branching machinery.

04 Perceive semantically

Driving a web application through its accessibility tree rather than its pixels or its DOM aligns the agent's representation with the application's declared semantics. The result is more robust than selectors and far cheaper than vision, and it produces logs a human can read.

05 Make failure legible, then trust the model with it

The framework never retries. It records failures faithfully and puts them in front of the model, which produces recovery within a goal and re-strategising across goals. Run 2 of the observed cycle is the whole argument in miniature: a failure inside a goal became information for the planner, and the case closed correctly.

06 Procedure is data, invariants are prompt

Case-specific procedure lives in a document an operations owner can edit. Cross-cutting safety rules (gateways, distinctions, evidence requirements, outcome consistency) live in the prompts, expressed with the negative examples that motivated them.

10

Maturity and roadmap

The observed cycle is small, but it is live and it exercises every terminal branch, and several properties usually asserted about agent systems are here directly evidenced by the reports. Four materially different outcomes came out of the same three components with no branch-specific code anywhere in the system; branching is not implemented, it is what the planner does when it reads a goal summary and consults the guide, so a fifth outcome is an edit to a prose document rather than a change to the graph. Across 95 tool calls nothing was fabricated in either direction: fields the email lacked were reported absent, fields it carried survived beside them, and both coverage end dates were read from policy detail pages rather than inferred from list views. The safety checks held even under the session-reuse shortcut designed to skip one of them: runs 2 and 3 saved their logins and still ran the

gateway and the expiry comparison in full. The one failed goal in the cycle was a correct negative, reported honestly and converted by the planner into the right customer outcome. And every run closed on the planner's own `done` signal well inside its budgets (at most seven goals of ten, at most fifteen recorded actions of twenty-five) with no operator intervention and no framework-level retries.

Four extensions follow naturally, all of them capability work rather than repair, and none affecting the outcomes in Section 7.

NEXT

Richer outcome reporting

Resolution is the conjunction of goal states, so run 2 is recorded as unresolved on the strength of the diagnostic goal that produced a correct rejection. Separating goal-level failure from case-level disposition (one predicate in the coordinator) would let it report as resolved with a rejection outcome.

NEXT

Parallel claim processing

Cases are already isolated from one another, each with its own state object and report, so moving from a shared browser page to a context per claim is a deployment change rather than a redesign.

NEXT

Checkpointing and approval gates

The graph runtime supports checkpointing, which would let a case survive a restart and, more valuably, allow an operator to sign off at the two points that warrant review: before a claim form is submitted and before an outbound reply is sent. The coordinator is already where every transition passes, so it is already the natural place to suspend.

NEXT

A regression corpus from the reports

A report is a complete, replayable description of a run, so a corpus of recorded cases plus the goals each branch should produce turns into a regression suite that catches prompt changes before they reach a mailbox, and would have flagged the one behavioural inconsistency in the cycle, run 5 closing without the mark-as-read step that runs 4 and 6 both performed.

Smaller items are tracked and none affected the cycle reported here: the narrative log uses a fixed path rather than a timestamped one, the report schema counts an `ABANDONED` state no code path sets, a click that raises but leaves a readable page is reported as successful rather than uncertain, and the prose fallback for a worker that declines to call `goal_complete` (a path that never fired) detects failure by string match rather than structurally.

Conclusion

The Agent Execution Mesh treats cross-application workflow automation as a problem of separation rather than of model capability. It separates deciding what to do from deciding how to do it, and both from deciding when the work is finished; it separates the procedure, which changes per deployment, from the invariants, which do not; and it separates what each agent is permitted to see from what merely exists in the state.

The deployment bears the design out. A single implementation handled four materially different outcomes with no branch-specific code, because branching is what a planner does when it reads a summary and consults a guide. Twenty-four of twenty-five goals completed; the one that did not was a correct negative that the planner turned into the right customer response rather than an aborted run. Nothing was fabricated, in either direction: absent fields were reported absent, and present fields survived beside them. A gateway placed before the first expensive action rejected half the cases for roughly a tenth of the tool calls and left no partial records behind. And per-goal accounting surfaced a cost distribution nobody had predicted: writing the client note is more expensive than filing the claim it describes.

None of this required a larger model. It required deciding, for each question the system must answer, whether that question is genuinely ambiguous, and routing it to ordinary code when it is not. The parts of the problem that are ambiguous are handled by models with tightly controlled context; the parts that are not are handled by a hundred and fifty lines of Python that cannot drift, cannot hallucinate, and can be read in a single sitting. That division, rather than any property of the models themselves, is what makes the system's behaviour predictable enough to put in front of customers.

THE LESSON

None of this required a larger model.

