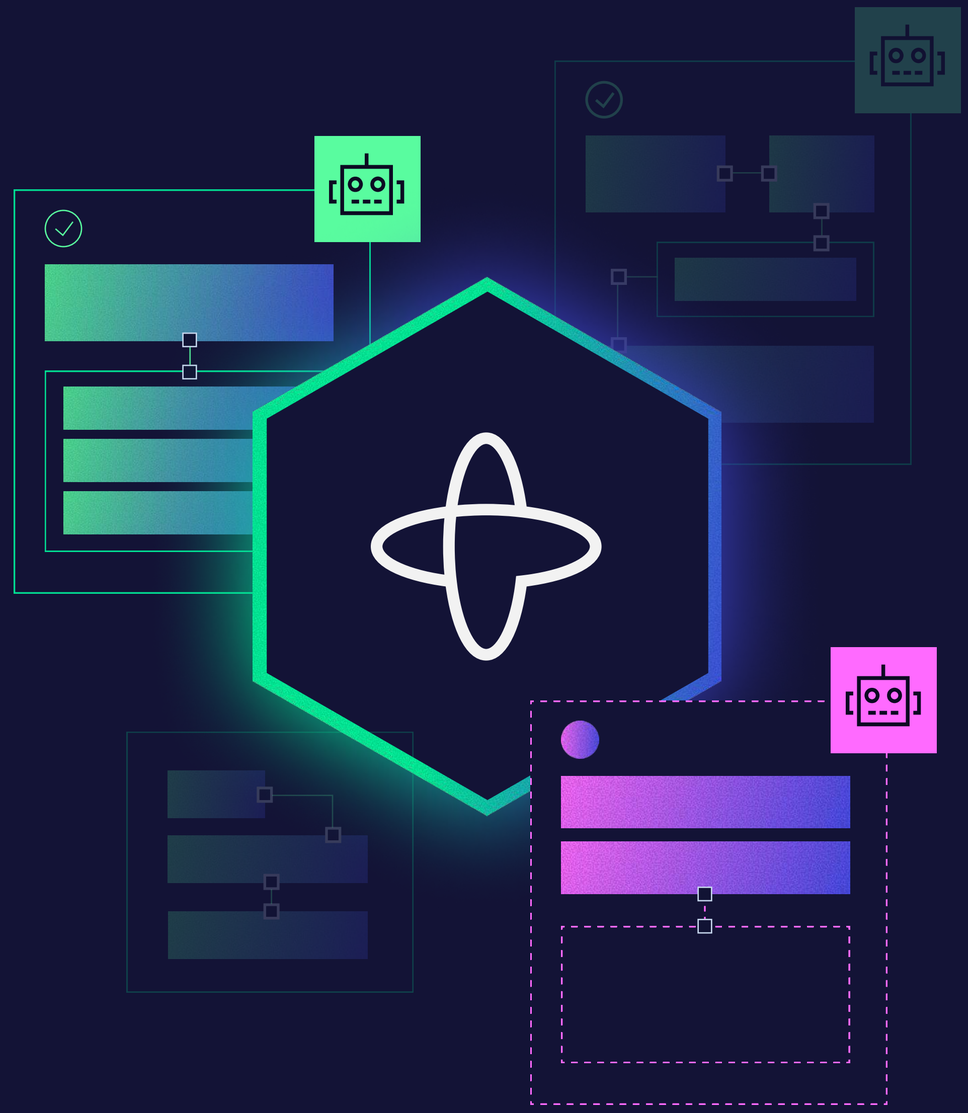
Durable orchestration for multi-agent AI systems

How to split an agent along real architectural boundaries, choose the right coordination models, and keep the result running through failure



Temporal

Table of contents

Introduction 04

When one agent becomes too much 06

- Evidence that a boundary already exists 08
- When the boundary is not a judgment call 09
- What a boundary costs 09
- Where the microservices analogy stops 10

Making the execution durable 11

- Keep orchestration in the Workflow and external calls in Activities 12
- Pick up where you left off 14
- Retry safely 16
- Wait without holding a Worker 17

How to split an agent 18
- Define the capability contract 18
- Keep the tool catalog behind the boundary 19
- Give the boundary one owner 21
- Decide how much of the flow needs an agent 21

Choosing the coordination boundary 21
- At a glance 23
- Start with Activities 23
- Use a Child Workflow for a subexecution 24
- Use Nexus for independent ownership 24
- Make the boundary operational 26

The Acme request, end to end 27

An implementation checklist 29

Conclusion 31
- Continue exploring 32

Agent systems often start with one prompt, one set of tools, and one loop. That is a sensible way to test an idea.

Trouble starts when every new capability arrives with more instructions, more tools, and more context, and when the first production incident reveals that the loop was never designed to be interrupted.

Those are two different problems, and they are usually solved in the wrong order. Teams reach for a multi-agent diagram when what they have is an execution problem, or they add a queue and a retry loop when what they have is an ownership problem. Natural-language instructions do not remove the need for boundaries, and a boundary that cannot survive a Worker restart is a diagram rather than an architecture.

This paper follows one customer-operations agent as it grows into a multi-agent system. It shows:

- How to tell a real boundary from a long prompt
- What Durable Execution provides once the boundaries are drawn
- How three Temporal primitives express three different relationships between capabilities: an Activity for one focused call, a Child Workflow for a subexecution you own, and a Nexus Operation for a capability another team runs
- How to prove the design works by failing it deliberately

When one agent becomes too much

Say you build a customer-operations agent to help an account team prepare for renewal meetings. It reads the CRM and product-usage data, uses approved public sources to fill gaps, and writes a briefing. After a few rounds of prompt and tool work, it can handle a request like this:

```
Prepare a briefing for the Acme renewal meeting. Summarize the account, identify the risks, and draft the follow-up we should send afterward.
```

01 **The first version has a narrow job.**
One team owns it, its tools cover one domain, and its output is one briefing.

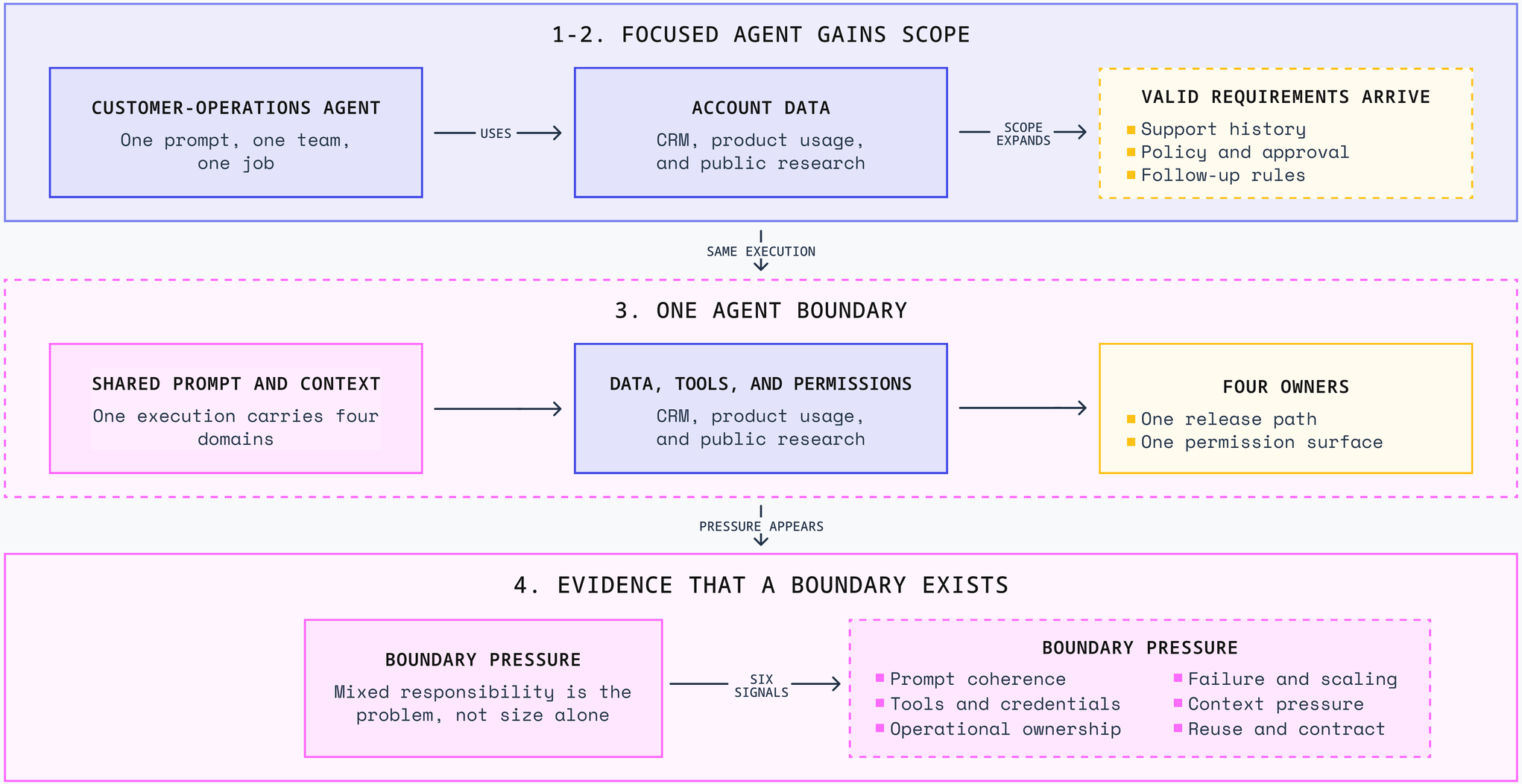
02 **Then it succeeds.**
Support asks it to include case history. Legal adds renewal-term rules and an approval path. The communications team contributes instructions for the follow-up. Each addition is reasonable on its own, so the team extends the same prompt and attaches another tool.

03 **The agent still runs, but its architecture has changed.**
Several teams now contribute instructions. The tool catalog spans CRM searches, product analytics, policy retrieval, approval actions, and messaging. A change to one area can alter behavior elsewhere because every instruction and schema competes inside the same context.

04 **Software teams have seen a version of this before.**
Starting with a monolithic application is often reasonable while the domain is still taking shape, and plenty of good systems stay that way for years. What separates a healthy monolith from a failing one is not size but internal structure: whether the parts have names, owners, and edges. Agent systems need the same distinction, and they need it earlier, because a prompt has no compiler to enforce it.

05 **Length is the other thing that changed, and it is easy to miss**
The first version answered in seconds. This one reads six systems, waits for a legal approval, and can stay open for two days. Every additional minute is more time in which a provider can rate-limit, a process can die, and a Worker can be recycled underneath the run.

Figure 1. How a focused agent becomes a monolith



Evidence that a boundary already exists

Read the two columns as opposing evidence rather than a scorecard. A single row rarely settles the question.

Keep one agent when…

- Its instructions describe one coherent job
- Its tools share a domain and permission model
- One team owns its quality and release cadence
- Its work shares a failure and scaling profile
- The full context helps the model decide
- The capability exists only for this request
- Everything it does sits in one permission domain

Extract a capability when…

- The work has its own domain language and decision rules
- It needs distinct tools, data, or credentials
- Another team needs to release and operate parts independently
- It needs different timeouts, capacity, or failure handling
- Unrelated instructions and schemas are crowding the context
- Other applications could reuse it behind a stable contract
- Policy or regulation requires two of its duties to stay separate

Prompt length is a symptom worth investigating, not a boundary in itself. A large agent may perform one complicated job. A short agent can still combine unrelated authority and ownership. The case for splitting gets stronger when several conditions in the right-hand column line up at once.

Until they do, the answer is structure rather than separation. Business rules that are already known can live in ordinary code instead of instructions, model and tool calls can sit behind narrow interfaces, and prompts can be assembled from modules with named owners. A team that does this defers the cost of a multi-agent system without deferring the architecture, which is the part that gets expensive to add later.

When the boundary is not a judgment call

Everything above is a trade-off. Separation of duties is not.
If one team may read customer records and another may approve commercial terms, an agent holding both capabilities holds a combination no single person in the organization is permitted to hold. Each permission was granted on its own. The union was never reviewed, because until the agent existed there was nothing to review it on. Auditors ask who can do what, and for a monolithic agent the honest answer is that anything in the tool catalog can be reached by anything in the prompt.

Compliance is a large topic and this paper alone will not settle it. The narrow point is that where policy separates two duties, the boundary has already been decided for you, and the only remaining question is which coordination model enforces it.

What a boundary costs

Splitting is not free, and the costs are easy to underestimate because they arrive later than the benefits. Every boundary adds a serialization step and a schema to version. It moves failure handling to the caller, which now has to decide what a partial result means. It makes end-to-end evals harder, because the quality of the whole answer is no longer produced in one place.

WATCH OUT

If most of the parent's context has to cross the boundary anyway, the split has added a contract without removing a dependency. That is the most common form of premature decomposition in agent systems, and it is easier to see before the extraction than after.

Where the microservices analogy stops

The comparison to service architecture is useful, and it stops being useful in four places worth naming.

01 **The interface uses a natural language,**
so it cannot be checked statically. A compiler will not tell you that a specialist quietly started interpreting `risk` differently.

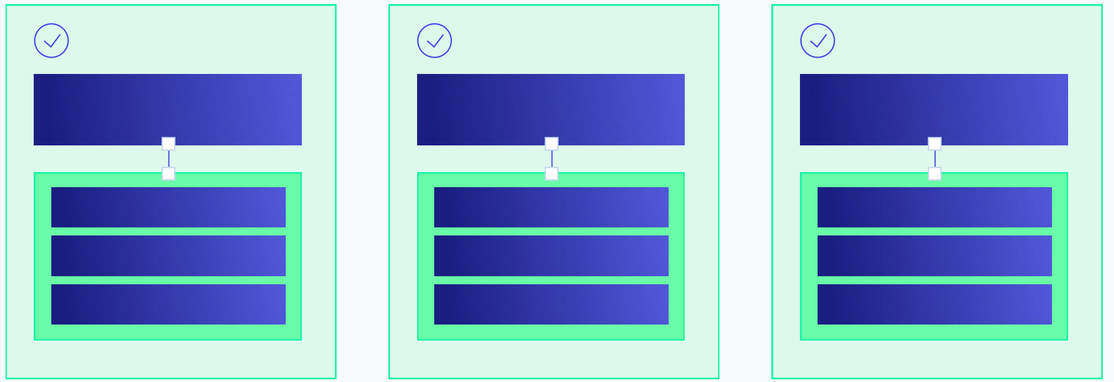
02 **Context is a contended, finite resource**
shared by everything inside one agent, which has no equivalent in a service call.

03 **Execution is non-deterministic,**
so the same input can take a different path on retry and produce a different, also-plausible answer.

04 **Failure is often silent:**
a service returns a `500`, while an agent returns a fluent paragraph that is wrong.

Those four differences argue for the same discipline as services, applied harder:

- Contracts should be explicit and typed where possible.
- Each capability should be evaluated on its own, not only through the finished output.
- Every step should leave a record you can read after the fact.



Making the execution durable

We know where this is going: more than one agent. Before drawing a single boundary, it is worth naming what the split will cost, because the industry already paid this bill once, moving from monoliths to microservices.

Breaking up one process into several introduces problems the single process never had: partial failure, retries, duplicate writes, work that never has to survive a restart on a machine that no longer exists. Agent systems inherit every one of those and bring their own: a step can take minutes, a human approval can take days, and each repeated model call arrives with an invoice.

The reflex is to defer all of that until the architecture is drawn. There are two reasons not to:

- **You cannot evaluate a multi-agent design that cannot finish**
  A run that dies halfway and restarts from nothing tells you only that it died. It tells you nothing about whether the boundaries you drew were the right ones, whether the specialist returned a usable brief, or whether the approval step belongs where you put it. Durability is what turns architecture into something you can actually measure.
- **This is the cheap part**
  Getting the boundaries wrong costs weeks of re-plumbing across teams. Getting durability is mostly a matter of putting calls in the right place, and the rest of this section is that one decision applied four times.

Here is what it looks like without durability in place.

01 **The Acme Agent Runs Into Production**

02 **A Model Provider Returns A Rate-Limit Error** After The CRM Lookup Has Already Completed

03 **A Worker Restarts** While The Request Is Waiting To Try Again

04 **A Stateless Process Starts Over:** It Repeats The Completed Research, Pays For The Same Model Call A Second Time, And, If The Follow-Up Had Already Been Sent, Sends It Twice

Keep orchestration in the Workflow and external calls in Activities

A Workflow holds the application logic: the sequence, the branching, and the decision about what happens next. Every call that leaves the process goes in an Activity: model calls, database reads, tool invocations over MCP or any other protocol, and outbound messages.

That split is what buys durability. The Temporal Service records the result of each Activity as it completes, so an interrupted run comes back with that work already done. The CRM lookup that finished before the crash stays finished. The model output you already paid for is still there.

It is also the rule most often broken by accident. Temporal's OpenAI Agents SDK integrations for Python and TypeScript apply it for you at the model boundary: the agent loop, tool selection, and handoffs run in Workflow code, while every model call is dispatched to an Activity.

python

```python
from datetime import timedelta

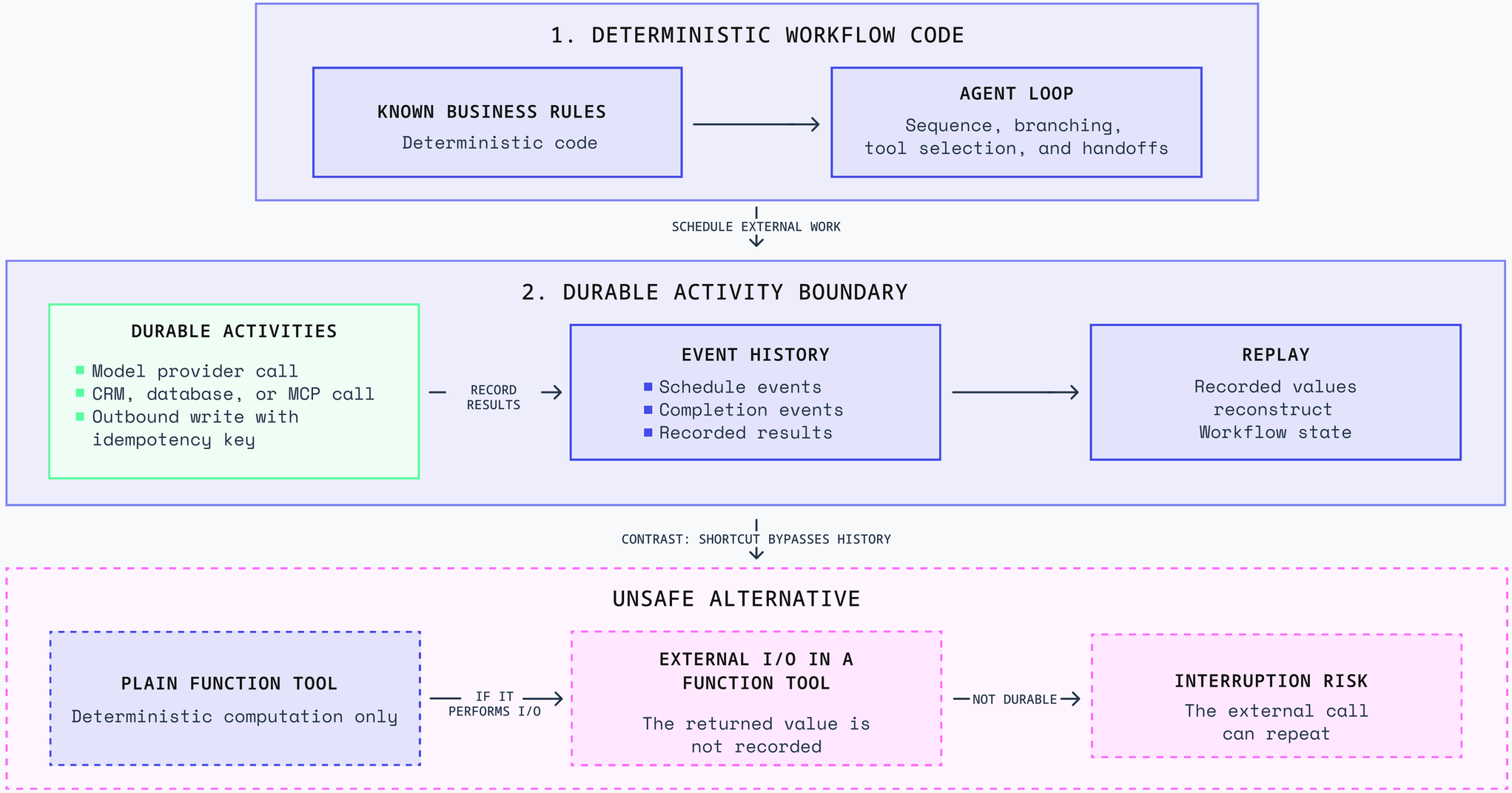
from agents import Agent, Runner
from temporalio import workflow
from temporalio.contrib.openai_agents import workflow as temporal_agents

@workflow.defn
class AccountIntelligenceAgent:
    @workflow.run
    async def run(self, account_id: str) -> str:
        agent = Agent(
            name="Account intelligence",
            instructions="Summarize the account and identify renewal risks.",
            tools=[
                # activity_as_tool puts this call on the far side of the
                # durable boundary: it performs I/O, so it runs as an Activity
                temporal_agents.activity_as_tool(
                    fetch_crm_account,
                    start_to_close_timeout=timedelta(seconds=30),
                ),
            ],
        )
        result = await Runner.run(agent, input=account_id)
        return result.final_output
```

✦ WATCH OUT

Model calls are routed through Activities automatically. Tools are not. A plain function tool runs inside the Workflow, where nothing records what it returned, so an interrupted run calls it again: the same search, the same charge, and if it wrote something, the same write. Any tool that touches an external system needs activity_as_tool or a Nexus Operation behind it. This is the most common source of a loop that looks durable and is not.

Figure 2. Where the durable boundary goes



Pick up where you left off

The Temporal Service records progress in Event History. When a Worker crashes, another Worker rebuilds the Workflow's state from that history and carries on from where the first one stopped. Completed Activity results are read from history rather than requested again, so recovery does not reissue the CRM lookup or repurchase the model output that preceded the failure.

Temporal primitive	Role in the Acme request
Namespace	The isolation unit for Workflows, Task Queues, access, and quotas; usually one per team or environment
Workflow	Owns the renewal-preparation request and its durable state
Worker	Any process polling a Task Queue; it runs Workflow and Activity code but holds no state of its own
Task Queue	The routing and scaling unit that decides which Workers pick up which work
Activity	Calls a model, reads an external system, or sends a message
Event History	Records the events needed to reconstruct Workflow state
Signal	Carries an approver's asynchronous decision into the Workflow
Heartbeat	Reports progress and checkpoint details from a long-running Activity

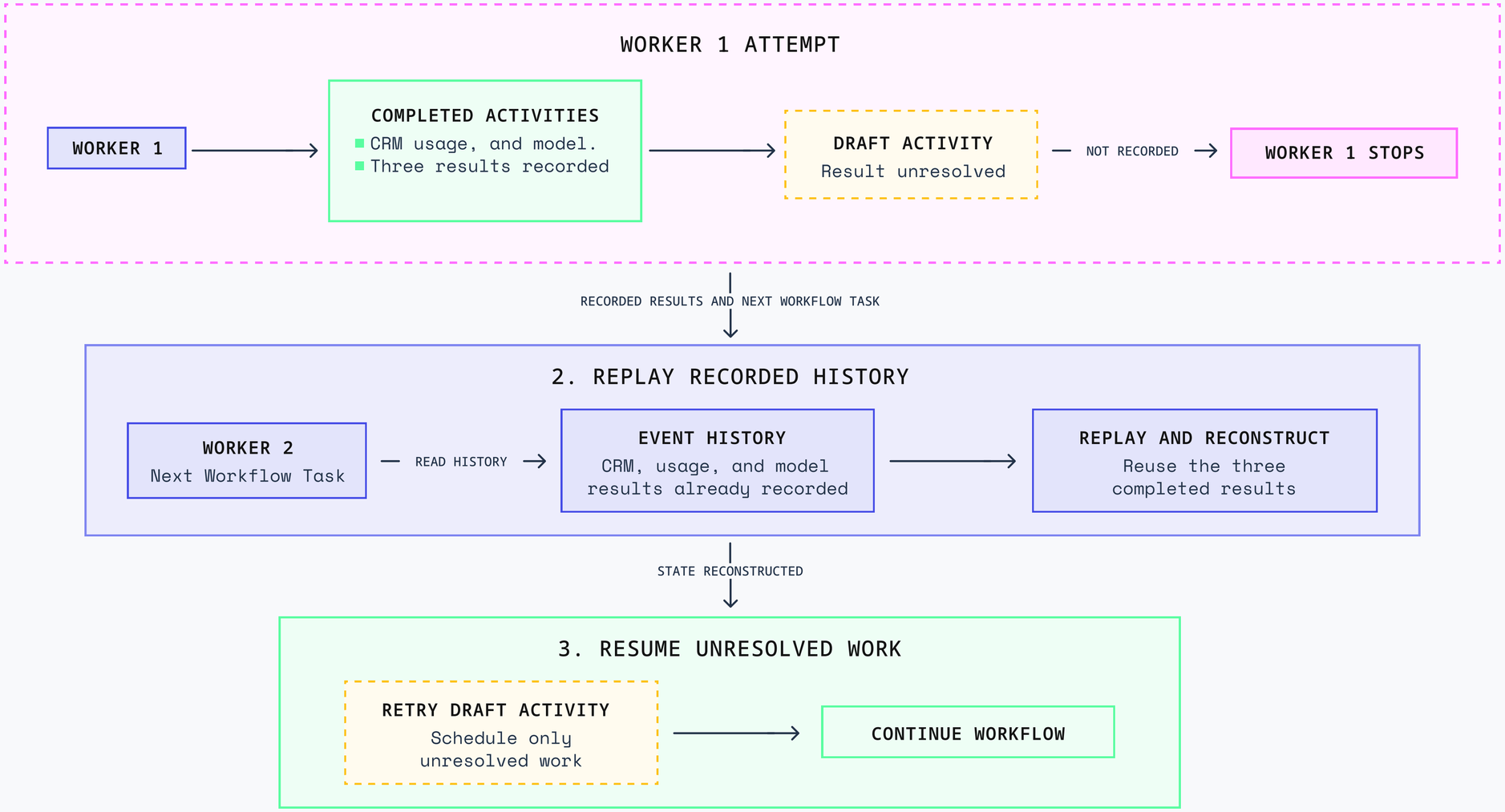
Event History is also the artifact you read after an incident. It shows which model call ran, with which inputs, in what order, and how many times, which is a stronger position than reconstructing an agent run from application logs.

WATCH OUT

Payloads have a size limit, and agent runs are unusually good at reaching it: long conversations, retrieved documents, whole research corpora.

Keep large values out of Activity arguments and return values, and pass a reference instead. External Storage does this for you, offloading payloads above a threshold to a store you control and passing a small token through the history.

Figure 3. Recovery through replay



Retry safely

Each Activity needs timeouts and a Retry Policy that match its dependency. A rate-limited model is worth retrying with backoff. A malformed request is not, and retrying it 20 times only turns a fast failure into a slow and expensive one.

`non_retryable_error_types` is the knob that separates the two: leaving it empty is how a validation bug becomes a bill.

python

```python
brief = await workflow.execute_activity(
    call_model,
    request,
    start_to_close_timeout=timedelta(seconds=60),
    retry_policy=RetryPolicy(
        initial interval=timedelta(seconds=2) ,
        backoff_coefficient=2.0,
        maximum_attempts=5,
        non_retryable_error_types=["InvalidRequest", "ContentPolicyViolation"],
    ),
)
```

Retries are at-least-once, not exactly-once.
An Activity attempt can finish an external write and then fail before the completion reaches the Temporal Service, and the retry will run the write again.

Give every side effect an idempotency key, a conditional Update, or another deduplication strategy, and derive the key deterministically from Workflow state, such as `workflow.info().workflow_id`, so that the same run always produces the same key:

python

```python
await workflow.execute_activity(
    send_follow_up,
    SendRequest(draft=draft, idempotency_key=f"{workflow.info().workflow_id}-follow-up"),
    start_to_close_timeout=timedelta(seconds=30),
)
```

✦ WATCH OUT
Audit the retries you did not configure. Model clients, HTTP libraries, and MCP adapters often retry internally, multiplying latency and provider spend underneath the Retry Policy you can see.

Wait without holding a Worker

Approvals can take hours or days, and no process should be held open for them. A Workflow can wait for a decision delivered as a Signal without keeping a Worker, a thread, or a connection attached to the request.

python

```
py
from datetime import timedelta
from typing import Optional

from temporalio import workflow


@workflow.defn
class CustomerOperationsWorkflow:
    def __init__(self) -> None:
        self._decision: Optional[ApprovalDecision] = None

    @workflow.run
    async def run(self) -> ApprovalDecision:
        # The policy review would happen here.
        return await self.await_approval()

    @workflow.signal
    def submit_approval(self, decision: ApprovalDecision) -> None:
        self._decision = decision

    async def await_approval(self) -> ApprovalDecision:
        await workflow.execute_activity(
            notify_approver,
            start_to_close_timeout=timedelta(seconds=30),
        )
        await workflow.wait_condition(
            lambda: self._decision is not None,
            timeout=timedelta(days=3),
        )

        assert self._decision is not None
        return self._decision
```

Because the wait runs in Workflow code, Temporal can reconstruct the Workflow from Event History without holding a process open. `workflow.wait_condition` blocks until the condition becomes true.

Use Heartbeats for the other kind of waiting: progress inside a single long-running Activity, such as a research pass over many sources. A retry can resume from the most recent Heartbeat details that reached the Temporal Service. Heartbeats are throttled, so treat that checkpoint as best-effort progress rather than a transaction boundary.

✦ IN THE ACME SYSTEM

Durable Execution is now doing three jobs:

1) completed research is never repurchased,
2) the approval can arrive the next morning without a process waiting for it,
3) the send step can be retried without sending twice. None of that required a second agent.

How to split an agent

With Durable Execution in place, the question becomes where to put the seams.
The microservices literature uses bounded context for a domain boundary within which a model and its language remain consistent. The term is useful here too. Account intelligence is one bounded context in the Acme system: it knows how account records, usage data, support history, and public research fit together. Commercial-policy review is another: it knows which rules apply and which exceptions require approval. **Writing the contract is how you find out whether the bounded context is real.**

Define the capability contract

Contract element	Account-intelligence decision
Responsibility	Build a sourced picture of the account and identify renewal risks
Input	`account_id`, as_of, and `meeting_date`
Output	Summary, categorized risks, supporting evidence, confidence, and source references
Failure behavior	Mark partial results when optional research fails; surface missing required systems as an error
Owner	Account-intelligence team, including its prompt, tools, evals, deployment, and on-call path

The coordinator now calls one capability, `build_account_brief`, instead of inheriting a prompt and a set of internal tools. The contract also gives the specialist team something concrete to evaluate: tests can assert required sources, allowed risk categories, and evidence fields without parsing a free-form essay.

Failure behavior belongs in that contract for the same reason it belongs in a service contract. The caller needs to know whether a partial result is usable, which errors are retryable, and which failures require a person.

Keep the tool catalog behind the boundary

MCP makes tool integration easier, and each connection widens an agent's scope — the instructions, tools, data, and credentials available to it. Scope is what a contract is supposed to contain.

Be concrete about where each connection lives. The CRM MCP server is configured on the account-intelligence Worker and nowhere else, so `build_account_brief` can reach customer records and the coordinator cannot. The usage-analytics connection sits in the same place; the public-research schemas never enter the coordination context because the coordinator never loads them. And the drafting step, which runs as a model Activity inside the coordinator's own Workflow, holds no CRM credential at all: it receives the facts the brief already established.

RULE

The reduction in context is the point, and so is the reduction in blast radius. A credential scoped to one specialist cannot be used by a prompt injected somewhere else in the system.

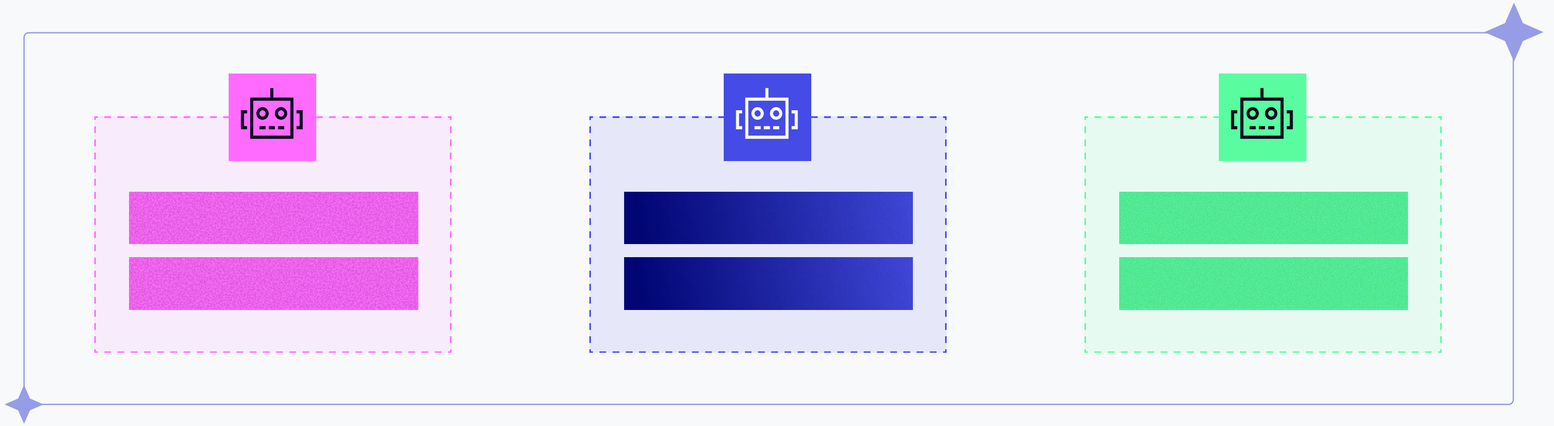
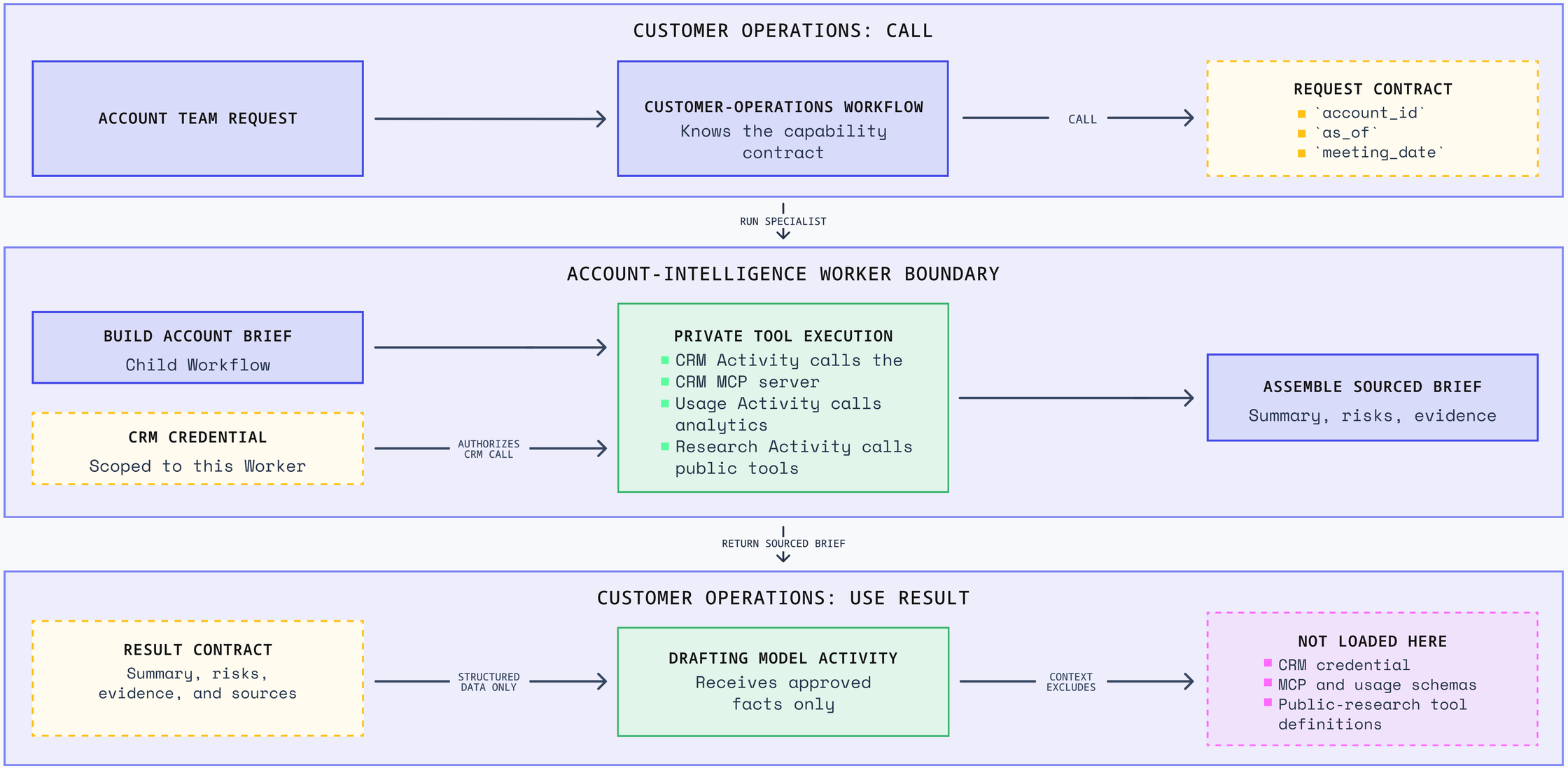


Figure 4. What the coordinator does not see



Give the boundary one owner

A small team should be able to own a capability's contract, prompt, tools, evals, deployment, and support. If every change still requires several teams to coordinate inside one shared agent, the split has renamed the problem rather than solved it. **Ownership is the test that survives when the others are ambiguous.**

Decide how much of the flow needs an agent

Splitting is one decision. How much of the result should be decided by a model is a separate one, and it is best to make this decision deliberately rather than inherit it from the prototype.

The Acme coordinator has a known top-level sequence:



Nothing about that order depends on what the model finds, so it belongs in ordinary Workflow code, where it is predictable, testable, and cheap to run. An agent loop earns its place one level down, inside account intelligence, where the research path genuinely varies by account and no fixed sequence would serve.

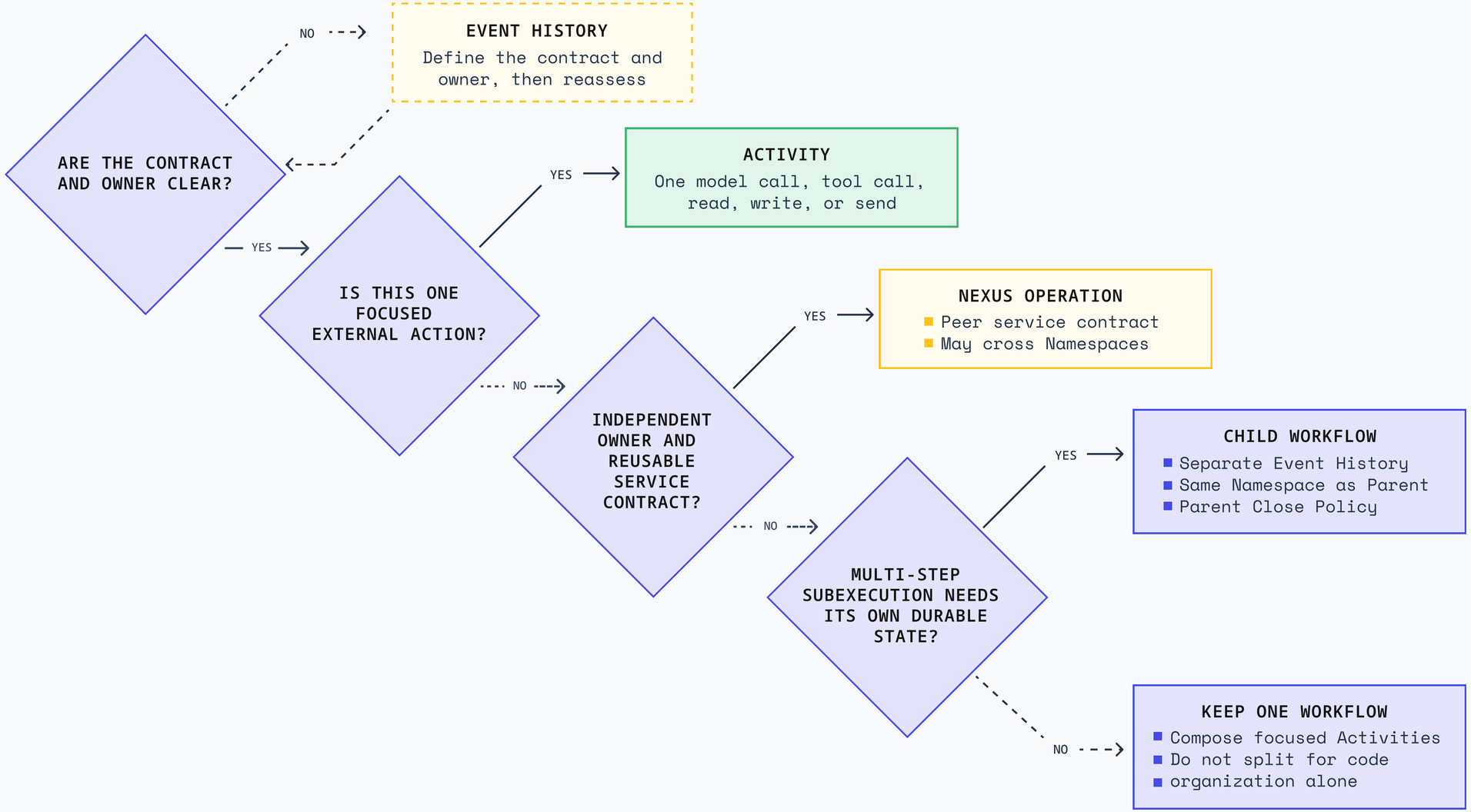
RULE

```
A fixed coordinator calling one open-ended specialist is
a common and underused shape. Reach for an agent loop at
the step that needs it, not at the top of the request.
```

Choosing the coordination boundary

Once a capability has a contract and owner, choose the narrowest Temporal primitive that matches its ownership and lifecycle. Network distance is a poor guide: a function on another Worker can still belong to the same application, while two capabilities in the same repository may need separate operational ownership.

Figure 5. Choosing the coordination boundary



At a glance

Consideration	Activities in one Workflow	Child Workflow	Nexus Operation
Relationship	Internal steps in one execution	Parent and Child	Peer caller and handler
Namespace	One Namespace	Same Namespace as the Parent	Can connect isolated Namespaces
State	Parent owns the composite state	Child has its own state and Event History	Handler owns its execution behind a service contract
Ownership	One application team	One application or closely coordinated teams	Independently operated application or team
Lifecycle	Same request lifecycle	Tied to the Parent through the Parent Close Policy	Independent lifecycle behind an Operation
Typical fit	Model calls, tool calls, or a focused specialist	A multi-step subexecution within a larger application	A reusable capability with its own owner

Start with Activities

Choose an Activity when the capability is one focused action in the outside world: a model call, a database query, a tool invocation or an outbound message. Activities can already run on their own Task Queues, so a focused communications step does not need its own Workflow merely because it uses a separate prompt.

Recommendation

Start from a single Workflow Definition while the number of Activity Executions and Signals stays bounded, and when in doubt, use an Activity.

Use a Child Workflow for a subexecution

A Child Workflow earns its place when a substantial part of the application needs its own durable state and failure handling. It gets a separate Event History and can run on its own Task Queue and Worker fleet, while remaining in the Parent Workflow's Namespace, with its lifecycle tied to the Parent through the Parent Close Policy. The `task_queue` argument routes the Child Workflow to the queue polled by that Worker fleet. Without it, the Child inherits the Parent's Task Queue.

python

```python
brief = await workflow.execute_child_workflow(
    AccountIntelligenceWorkflow.run,
    AccountBriefRequest(account_id=account_id, meeting_date=meeting_date),
    id=f"account-brief-{account_id}-{meeting_date}",
    task_queue="account-intelligence",       # route to the dedicated Worker fleet
    execution_timeout=timedelta(minutes=30),
)
```

Use Nexus for independent ownership

Temporal Nexus expresses a different relationship. A team exposes a named Service containing Nexus Operations, and a Nexus Endpoint, referenced by name as `commercial-policy-endpoint`, routes requests to the handler's Namespace and Task Queue. Callers depend on the Service contract, not on the implementation, and the two sides are peers rather than parent and child. That fits a commercial-policy capability owned by compliance and reused by several applications.

python

```python
policy = workflow.create_nexus_client(
    service=CommercialPolicyService,
    endpoint="commercial-policy-endpoint",
)
decision = await policy.execute_operation(
    CommercialPolicyService.review_renewal,
    ReviewRequest(account_id=account_id, proposal=proposal),
    schedule_to_close_timeout=timedelta(minutes=2),
)
```

RULE

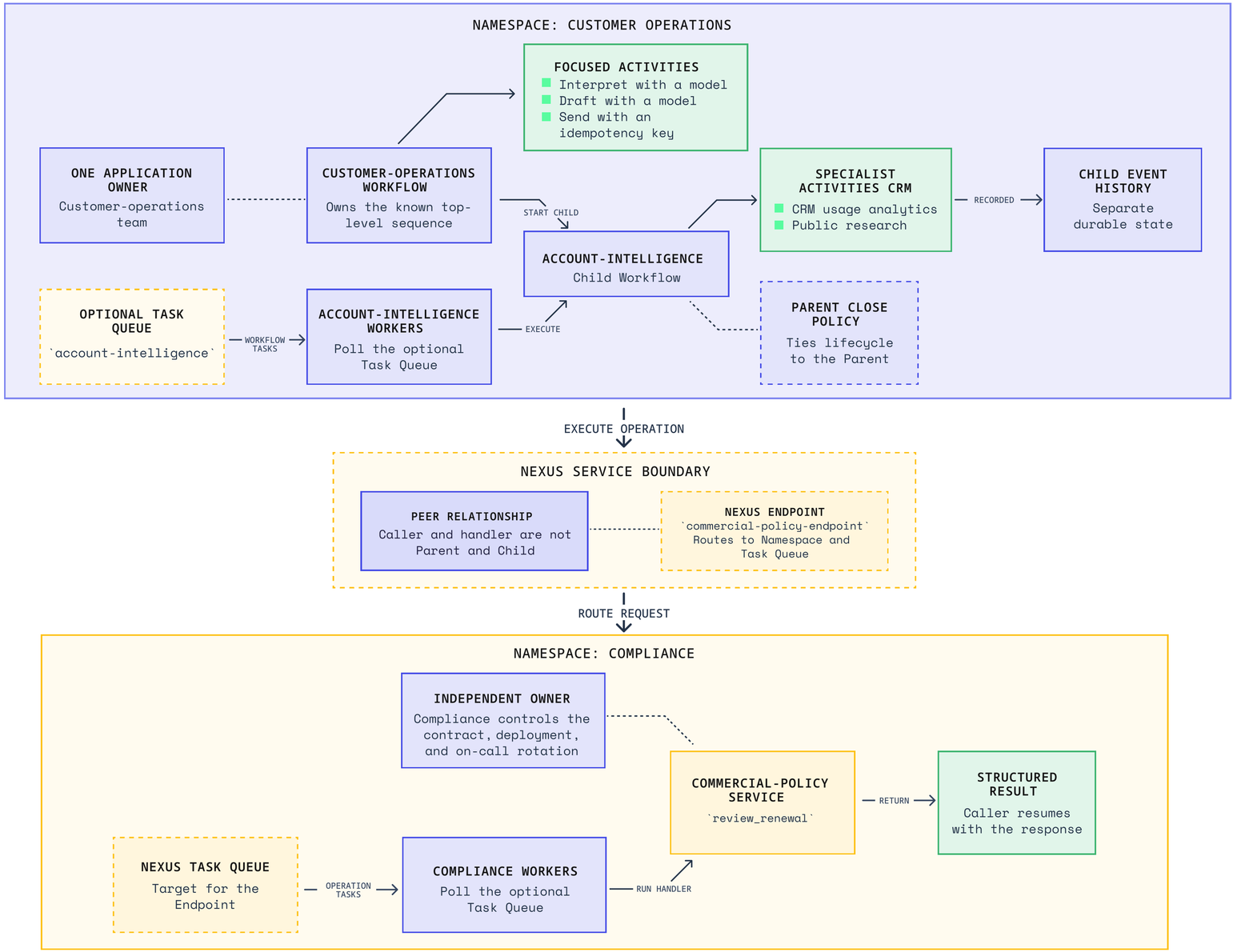
Nexus asks for two things in return: an Endpoint to register and a service contract to version across teams. That is a fair trade, and it is the only one of the three boundaries that buys independent ownership, an independent deployment cadence, and a Namespace boundary you can put an access policy on.

Take the trade when you need those. When the same group owns both sides of the call, a Child Workflow will usually do.

Nexus Operations are delivered at least once, exactly like Activity retries, so a handler that writes to an external system needs the same idempotency strategy described earlier. The boundary changes who owns the code, not what the code owes the caller.

An agent framework can point a tool at either kind of boundary. In the Python integration, `activity_as_tool` backs a tool with an Activity, and `nexus_operation_as_tool` lets an agent call a capability in another Namespace. The framework expresses the boundary; it does not choose it.

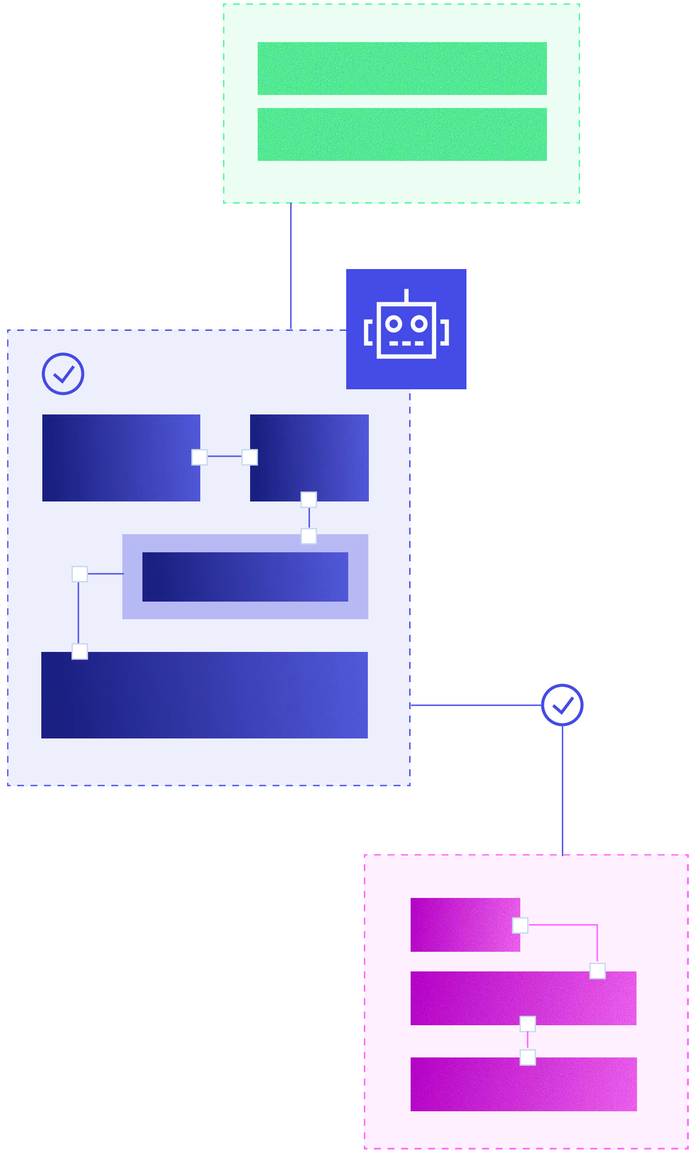
Figure 6. Three coordination boundaries in one system



Make the boundary operational

A logical specialist is not automatically a bulkhead. If every specialist shares one Worker process, Task Queue, concurrency limit, and credential, a traffic spike or a provider outage still takes down the whole system. Every reason for a boundary has a control that enforces it, and **a boundary without its control is decoration.**

The reason you drew the boundary	The control that makes the boundary real
Different scaling or deployment cadence	Separate Task Queues and Worker fleets
Different permissions	Credentials scoped to the specialist's tools and data
Different failure profile	Capability-specific timeouts, Retry Policies, rate limits, and concurrency limits
Independent security or fault isolation	Separate Namespaces, reached through a Nexus Endpoint



The Acme request, end to end

01

Accept the request

The customer-operations Workflow receives the renewal request under a stable Workflow ID. Workflow code drives the known business sequence, while a model Activity interprets the free-form instruction. The coordinator holds the account-intelligence contract without loading CRM, analytics, or search schemas into its context.

02

Build the account brief

The coordinator starts account intelligence as a Child Workflow. This is a multi-step capability: it queries the CRM and usage systems, may fan out public research, applies account-health rules, and returns a sourced brief. The same group owns both Workflows, but the subexecution gets its own state, Retry Policies, Event History, and Worker capacity.

03

Review the proposed action

The coordinator sends the brief and renewal proposal to a commercial-policy Nexus Operation. Compliance owns that capability in another Namespace and exposes it to several applications. The Operation returns an allowed, denied, or approval-required decision with the applicable rule.

04

Wait for a person

Acme's offer requires approval, so the Workflow records the policy result and waits. An Activity notifies the approver, and the decision returns as a Signal. It arrives the next morning, long after the Worker that started the request has been replaced.

05

Draft and send the follow-up

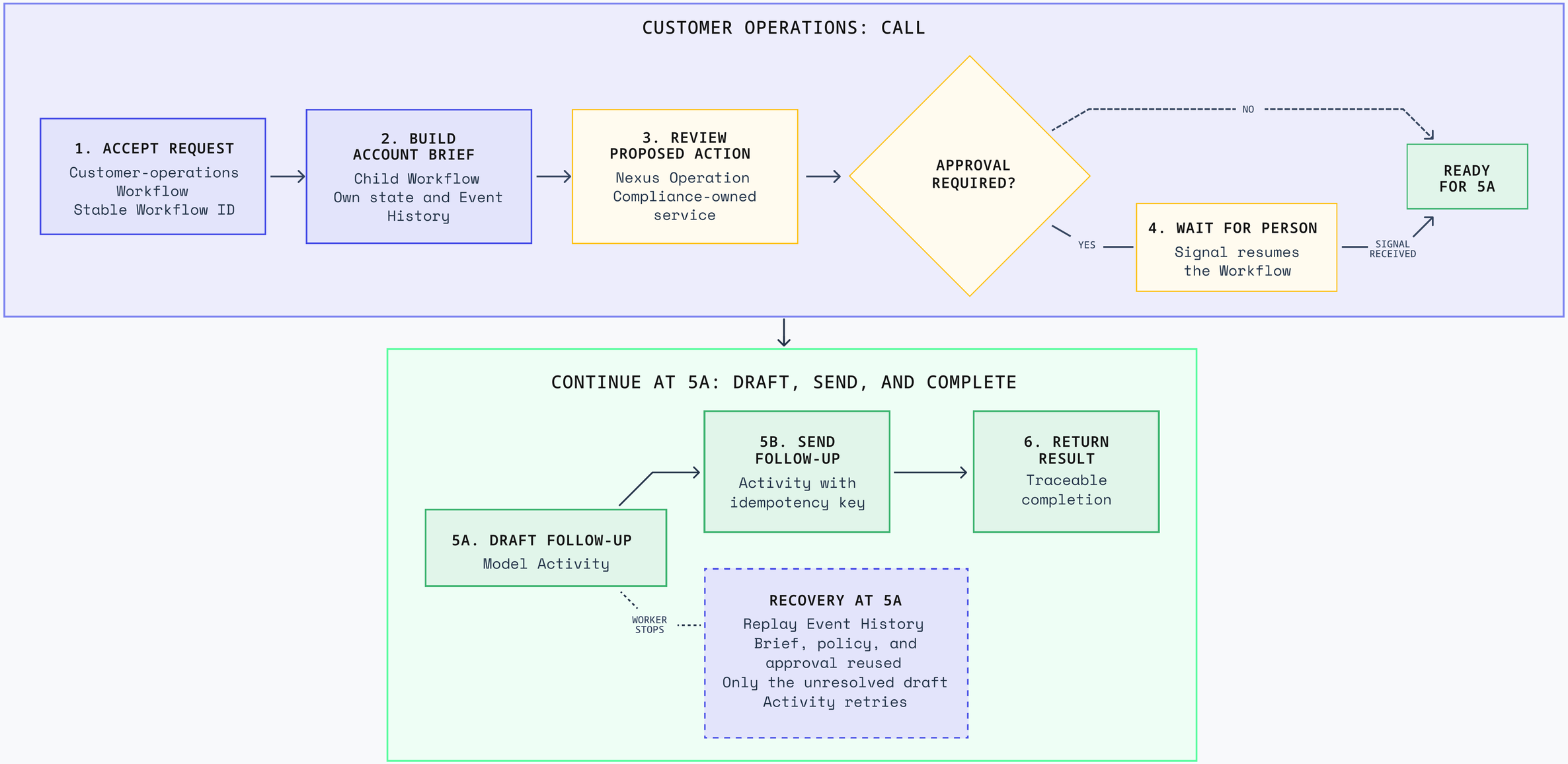
Drafting stays a model Activity, because it is one focused call owned by the customer-operations team. It receives approved facts without CRM credentials or the policy service's internal context. Sending is a separate Activity with an idempotency key, so retrying the send cannot duplicate the outbound message.

06

Complete with a traceable result

The brief, policy decision, approval, draft, and send result all return through the customer-operations Workflow. When a Worker fails during drafting, replay restores the recorded research and approval and schedules only the unresolved work.

Figure 7. The Acme request, end to end



An implementation checklist

Do this...	You are done when...
☑ **Map responsibilities:** List the decisions the agent makes, the tools and data it can reach, the state it can change, and the team that owns each item	You can name the owner of every tool and every instruction block
☑ **Write the contract:** For each candidate capability, specify responsibility, input, structured output, failure behavior, and owner	A test can assert on the output without parsing prose, and the caller knows which errors are retryable
☑ **Choose the level:** Apply Figure 5. An Activity for one focused call, a Child Workflow for an owned subexecution, a Nexus Operation for another team's service	The reason for the boundary is written down next to the code
☑ **Audit the tool definitions:** Confirm every tool that performs I/O is backed by an Activity or a Nexus Operation, not a plain function tool	A search of tool definitions finds no tool that performs I/O without an Activity or Nexus Operation
☑ **Enforce the isolation:** Apply the control that matches each reason: Task Queues, scoped credentials, per-capability limits, or Namespaces	Saturating one specialist leaves the others serving traffic
☑ **Set failure policies:** Give each dependency timeouts and a Retry Policy, mark non-retryable errors explicitly, and disable internal retries in model and HTTP clients	Provider spend and worst-case latency are both bounded and predictable
☑ **Protect side effects:** Give every external write an idempotency strategy derived from durable Workflow state	Replaying a completed write produces no second write
☑ **Break it on purpose:** Kill a Worker after several Activities complete. Repeat during an approval wait, and after an external system accepts a write but loses the response	Recovery reuses recorded results, the delayed Signal still lands, and nothing is sent twice
☑ **Read the history:** Inspect Event History in the Temporal UI or CLI and confirm model and I/O calls appear at the intended boundaries	You can reconstruct the Workflow's execution path without relying on application logs

RELEASE CHECK

A production candidate survives a Worker restart after completed model and tool calls, accepts a human Signal

that arrives days later, retries a transient dependency failure without duplicating a write, and keeps a failing specialist's problems inside that specialist's operational boundary.

CONCLUSION

Start with one agent while the responsibility is coherent. Give it internal structure while it is still small enough to restructure.

When context, permissions, scaling, reuse, or ownership reveal a real boundary, extract a specialist behind a narrow contract, and choose the coordination model that matches how the capability is owned and how long it lives.

Temporal should express the boundary the application already has, not invent a new one.

Durability is what makes those boundaries hold. An architecture diagram assumes every box completes; production does not. When orchestration is deterministic and external calls run in Activities, Event History lets the Acme request survive whichever Worker happens to be running it. Each capability can then be understood, tested, operated, and changed on its own, without asking one prompt to carry the entire architecture.

Continue exploring →

Explore Temporal's AI agent resources.

Read Temporal's Durable AI documentation and the AI cookbook for runnable agent examples.

Review the Workflow, Activity, Child Workflow, and Nexus documentation.

Start a Temporal Cloud trial with $150 worth of free credits and test the failure path of an agent you already run.