AGENT ORCHESTRATION · DECISION GUIDE · JULY 2026

Loop, graph, or neither

Three ways to run an agent past the edge of one context window. Two of them are usually wrong for the task in front of you. This is how to tell which.

Current as of **26 July 2026**. Every claim is sourced at the end. Where the evidence is field reports rather than experiments, it says so.

THE WHOLE GUIDE IN ONE LINE

A loop lets the model choose the next step. A graph decides the next step before the model runs. Everything else follows from that.

Orchestration is not a sophistication ladder you climb. It is a choice about *when* control flow gets decided, and each answer bills you differently.

PART I · THE DEFAULT NOBODY PICKS

Most tasks need no orchestration at all

The cheapest architecture is the one you did not build.

Start here, because skipping this step is the single most expensive mistake in the category. Before choosing between a loop and a graph, establish that you need either one. The honest test is narrow: does the work genuinely exceed what one agent can hold in one context window, end to end?

For most tasks the answer is no, and a single session is not a compromise. It is the correct design. One session keeps the entire transcript available, so the model can see its own reasoning from twenty turns ago. It keeps a stable prompt prefix, so caching works without any effort on your part. It has no handoff, which means no place for state to get lost. Every orchestration pattern in this guide is a way of buying past the context limit, and each one pays for that with a loss of continuity.

The community mood has shifted toward this position. The dominant sentiment in practitioner forums through 2026 has been some version of *when not to build an agent*, and it is not contrarian posturing. Carnegie Mellon's agent benchmark work found the best-performing agent succeeded at roughly 30% of realistic office tasks, and chaining three agents together produced an end-to-end success rate near 34%.[1] Those numbers are not an argument against agents. They are an argument against adding coordination to a component that is not yet reliable on its own.

THE TEST

If you cannot state, in one sentence, what specifically overflows the context window, you do not have an orchestration problem. You have a prompt you have not tightened yet.

PART II · THE FIRST REAL OPTION

What a loop actually buys you

Fresh context every turn, at a price the one-line version never quotes.

A loop runs the agent repeatedly, giving it a clean context each iteration and keeping durable state on disk. The purest form is a shell one-liner that pipes a prompt file into a coding agent forever. The technique circulating under the name Ralph loop comes from Geoffrey Huntley, who was refreshingly honest about it: the loop is, in his words, deterministically bad in an undeterministic world, and building with it requires a belief in eventual consistency.

The foundation under the technique is solid and measured. Models degrade as input grows, and not only when the window fills. The canonical result is *Lost in the Middle*, which found a U-shaped accuracy curve as relevant information was slid through different positions in a long input: best at the very start or the very end, worst buried in the middle.[2] Chroma's *Context Rot* work tested 18 frontier models and found consistent degradation as input length grew, well before the window was full.[3] An agent that searches, explores, and backtracks for an hour accumulates noise, and that noise makes every later decision worse.

So a clean context per turn is a real benefit. The cost is the part that gets left out of the reposts, and it is severe.

The cache is the whole economy

Prompt caching only works on an exact prefix match. Anthropic's documentation is explicit that cache hits require identical prompt segments up to the cached block, and advises putting stable content at the very beginning.[4] OpenAI's caching behaves the same way, on an exact prefix match for prompts over 1,024 tokens.[5] A cache read costs roughly a tenth of the normal input price.

Now look at what a naive loop does to that. If every iteration rebuilds the prompt from scratch, reads a different set of files, and reconstructs context in a different order, the prefix differs every time and the cache never hits. You pay full price on a large prompt, hundreds of times. This is not a rounding error. It is the difference between a loop that is merely wasteful and one that is ruinous, and OpenAI's own documentation warns about exactly this interaction with context-engineering techniques that rewrite the prompt.

The disciplined version fixes it by construction. The system prompt, the spec, and the standing instructions go at the front and stay byte-identical across iterations, so they cache. Only the volatile tail varies. That is a deliberate design decision, and it is the line between the two versions of this pattern.

WHAT SURVIVES, WHAT DOES NOT

Files are a good place to keep state and a bad place to keep reasoning. The next iteration reads the artifact and sees *what* was done. It cannot see *why*, which leaves it free to undo a deliberate decision or re-litigate a settled question. If a later step depends on earlier reasoning, make the agent write the reasoning down as an artifact, not just the result.

Vendors converged on the shape of this independently, which is the strongest evidence the core idea is sound. Anthropic's writing on harnesses for long-running agents describes the problem precisely, as a project staffed by engineers working in shifts where each arrives with no memory of the previous shift, and their finding is blunt: even a frontier model looping across multiple context windows falls short of building a production-quality application when given only a high-level prompt.[6] The naive loop does not work. Their fix is structural, an initializer agent that sets up the environment once and a coding agent that makes incremental progress while leaving clear artifacts behind.

PART III · THE 2026 TURN

When a loop becomes a graph

The question that keeps getting asked, and the answer that resolves it.

In late July 2026 a short post about drawing agent workflows as graphs and handing the drawing to a coding agent went past 700,000 views in two days. The recipe was three lines long: draw a graph in any tool, even on paper, send it to the model, and ask it to write a script that implements the workflow. The most-liked reply was a single question, and nobody answered it cleanly.[7]

How is this not a loop?

It is a fair question, and the answer is precise. A loop *is* a graph. It is a graph with one node and an edge pointing back at itself, where the model decides at runtime whether to take that edge again. The difference between the two patterns is not topology. It is **when control flow gets decided**.

In a loop, the model chooses the next step during inference, every single iteration. You are paying tokens for routing decisions. In a graph, you decided the routing when you drew it, and the model only does the work inside each node. The edges are code. That single shift moves a recurring inference cost into a one-time authoring cost, and it is the entire economic argument for graphs.

1

One session

Control flow decided by the model, continuously, with full history visible. No handoff. Caches trivially. Rots as the transcript fills.

2

Loop

Control flow decided by the model each iteration, with history discarded and state on disk. Buys a clean context. Pays in routing tokens and lost reasoning.

3

Graph

Control flow decided by you, at authoring time, encoded as a script. Nodes are model calls, edges are code. Deterministic, inspectable, parallelisable. Pays in rigidity.

Read that ladder as a trade, not a ranking. Moving down it, you buy determinism and give up adaptivity. A graph cannot discover a step you did not draw. If the shape of the work is genuinely unknown until the agent looks at the results, a graph will either fail or quietly route around the thing you needed it to notice. That is the failure mode people hit when they graph-max a problem that was never a graph.

THE PRACTICAL VERSION

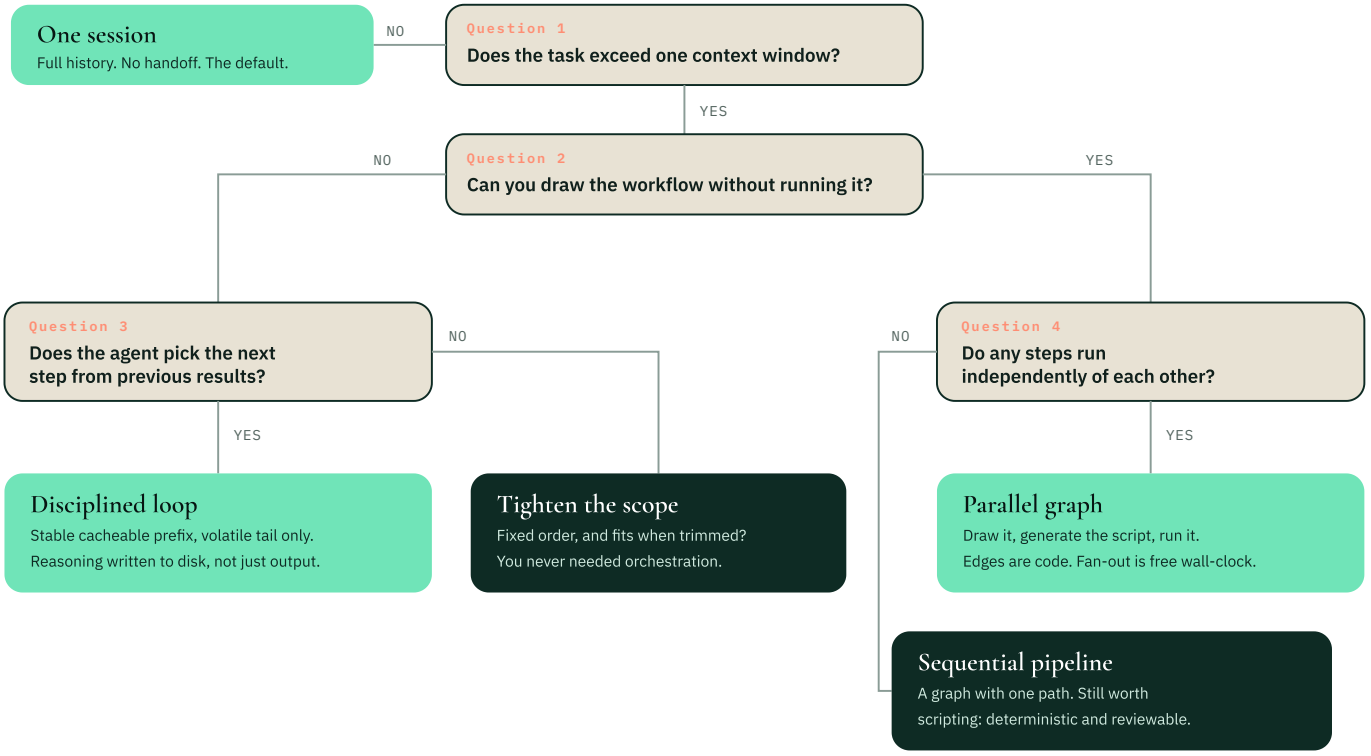
Graphs are worth reaching for when you can draw the workflow without running it. If drawing it requires knowing what step three returns, it is a loop. If you can draw it on paper over coffee, it is a graph, and generating the script from the drawing is now genuinely a two-step process.

There is a second, quieter benefit that the viral version undersells. A drawn graph is a review artifact. A colleague can look at it and say the retry edge is missing, or that two nodes should run in parallel, without reading a line of code or a single transcript. Loops do not produce anything a person can review at a glance. Their behaviour only exists in the run.

PART IV · THE DECISION

Four questions, in order

Answer them top to bottom. Stop at the first one that lands.



The tree encodes one opinion worth stating plainly. Question 2 comes before question 3 on purpose. Most teams ask *should this be a loop or a graph* as though it were a matter of taste, when the deciding fact is simply whether the workflow can be drawn in advance. If it can, drawing it is cheaper than every alternative. If it cannot, no amount of graph tooling will make the shape knowable, and you are in loop territory whether you like it or not.

PATTERN	WHO PICKS THE NEXT STEP	CACHE BEHAVIOUR	FAILS WHEN
One session	Model, with full history visible	Prefix stable, caches well by default	Transcript fills with noise and accuracy decays
Naive loop	Model, each iteration, context rebuilt	Prefix changes, cache misses, full price every loop	Token bill compounds and reasoning is lost
Disciplined loop	Model, each iteration, stable spec prefix	Spec caches, only the changed tail is uncached	The "why" is lost unless artifacts encode it
Graph	You, at authoring time	Per-node prefixes are stable and cache cleanly	The work needs a step you did not draw

PART V · THE BILL

Where the extra tokens go

Orchestration is not free, and the invoice arrives in a shape most teams do not forecast.

Every pattern above spends tokens on something other than the work itself. Naming those line items is what lets you predict the bill instead of discovering it.

Routing tokens. In a loop, the model re-reads enough state to decide what to do next, every iteration. That is real input cost spent on a decision, not on output. A graph pays this once, when you draw it. Over a few hundred iterations the difference stops being academic.

Re-grounding tokens. Each fresh context has to be told what it is doing. Spec, conventions, current state. In a disciplined loop this is exactly the part you keep byte-identical so it caches at roughly a tenth of the price. In a naive loop it is the part you pay full freight for, repeatedly, and it is usually the largest single block in the prompt.

Coordination overhead. Multi-agent structures multiply token consumption substantially compared with a single agent on the same task, because context gets duplicated across agents and results get passed between them. This is the cost that makes the reliability numbers in Part I bite: you are paying considerably more to run a structure whose end-to-end success rate is lower than its individual components.

Retry and repair. The one nobody budgets. When a handoff loses information, the next node produces something subtly wrong, and either a human catches it or a later node does. Both paths cost more than getting it right once. This is why the reasoning-on-disk discipline is an economic measure and not a tidiness one.

A FORECAST YOU CAN ACTUALLY RUN

Before building anything: estimate iterations, multiply by the re-grounding block size, and check whether that block is cacheable as written. If the prefix is not stable, that number is your floor and you are paying it in full. Teams are usually surprised by the magnitude, and the surprise is what kills projects mid-quarter.

PART VI · RECOGNISING TROUBLE

Four failure modes, and what each one looks like

Each pattern breaks in a characteristic way. Knowing the shape shortens the diagnosis.

Premature completion

A later iteration looks around, sees that progress was made, and declares the job finished when it is not. Anthropic names this explicitly as one of two failure modes a naive loop hits.[6] The tell is a run that terminates early and cleanly with an confident summary that does not match the artifacts. The fix is an explicit completion criterion checked against the spec, not against the agent's own judgement.

Half-built handoff

The other named failure. An iteration runs out of context mid-implementation and leaves the next one a partial feature with no explanation, and the next agent guesses at what happened. The tell is code that looks like two different authors disagreed. The fix is a checkpoint discipline where an iteration either completes a unit of work or explicitly records that it did not.

Silent re-litigation

The subtlest one, and the reason files-as-memory is not sufficient. A fresh instance reads the artifacts, does not see the deliberation that produced them, and undoes a deliberate decision because nothing recorded that it was deliberate. The tell is oscillation: the same file flipping between two approaches across iterations. The fix is writing decisions and their rationale as artifacts.

The undrawable edge

The graph-specific one. The workflow needed a step or a branch that was not in the drawing, so the run either fails at that point or routes around it and produces something plausible and wrong. The tell is a graph that succeeds on every test case you designed and fails on the first real input. The fix is not a bigger graph. It is recognising that this problem belonged in loop territory.

PART VII · BEFORE YOU BUILD

The one-page checklist

If you take one thing from this guide, take this page.

ESTABLISH YOU NEED ORCHESTRATION

- State in one sentence what specifically overflows the context window.
- Confirm a tightened single-session prompt genuinely cannot do it.
- Confirm the individual agent is reliable before adding coordination to it.

IF YOU ARE DRAWING A GRAPH

- Draw the whole workflow before writing any code, on paper is fine.
- Have someone else read the drawing and name a missing edge.
- Mark which nodes are independent, because that is your parallelism.
- Decide what happens on a node failure before you generate the script.

IF YOU ARE RUNNING A LOOP

- Put the system prompt, spec, and standing instructions first, byte-identical every iteration.
- Verify a cache hit on iteration two before running iteration three hundred.
- Make the agent write reasoning to disk, not only results.
- Define a completion criterion checked against the spec, not agent judgement.
- Set a hard iteration and token ceiling. Loops do not stop themselves.

EITHER WAY

- Forecast the token bill before building, using iterations times re-grounding block size.
- Instrument per-iteration cost so the compounding is visible on day one, not at month end.
- Keep the drawing or the spec in version control next to the code it generated.

THE CLOSING POSITION

Do not ask which pattern is most advanced. Ask whether you can draw the work before you run it.

If you can, draw it. If you cannot, loop carefully and keep the prefix stable. If neither question applies, you probably needed one session and a better prompt.

SOURCES

Where each claim comes from

Two of these are field reports rather than experiments, and are marked as such.

1. Carnegie Mellon University, TheAgentCompany benchmark. Best-performing agent completed approximately 30% of realistic office tasks; multi-agent chains degrade end-to-end. `arxiv.org/abs/2412.14161`
2. Liu et al., *Lost in the Middle: How Language Models Use Long Contexts*, 2023. U-shaped accuracy curve by position in long inputs. `arxiv.org/abs/2307.03172`
3. Chroma, *Context Rot*, July 2025. Degradation across 18 frontier models as input length grows, before the window fills. `trychroma.com/research/context-rot`
4. Anthropic, prompt caching documentation. Cache hits require identical prompt segments up to the cached block. `docs.anthropic.com/en/docs/build-with-claude/prompt-caching`
5. OpenAI, prompt caching guide. Exact prefix match for prompts over 1,024 tokens. `developers.openai.com/api/docs/guides/prompt-caching`
6. Anthropic, engineering writing on harnesses for long-running agents, November 2025. Names premature completion and half-built handoff as the two naive-loop failure modes. *Vendor engineering post, not a controlled study.*
7. A. Kotliarskyi (@alex_frantic), X, 24 July 2026. The graph-max recipe and the reply thread that asks how it differs from a loop. *Field report. The 700,000+ view count indicates reach, not validity.*

ON THE EVIDENCE

Context rot and the value of external memory are measured results. The loop and graph recipes built on top of them are not. There is no published head-to-head of a disciplined loop against a compaction-based long session on cost and quality. Treat the patterns as convergent engineering with documented costs, not as proven method.

Sangam Pandey writes about production AI engineering at sangampandey.info. Longer treatments of the material here: Ralph loops are not about throwing away context and context as a cost lever.

Version 1.0 · 26 July 2026 · Corrections welcome.