Loop Design: The Anthropic Playbook for Agentic Systems

D. R. Halvorsen M. Okonkwo L. Petrova J. S. Almeida

Agentic Systems Group

Correspondence: loops@agentic-systems.org · Preprint. Working draft, v1.3.

Abstract

Most practitioners still operate large language model agents the way they operated chatbots: one prompt, one reply, a human in the loop on every turn. We argue this is a transitional posture, not an endpoint. As agents acquire the ability to discover their own work, isolate their changes, and check each other, the unit of engineering shifts from the prompt to the loop: a scheduled, self-driving cycle that the operator designs once and supervises rarely. We present a field playbook for designing such loops. We decompose a loop into five movements, Discovery, Handoff, Verification, Persistence, and Scheduling, and define five levels of operator autonomy from manual prompting to fully autonomous operation. We give a structural account of why an agent that grades its own work tends to inflate its score, and why an independent verifier, the "thing that can say no," is the single highest-leverage component in the design. We close with a fourteen-step adoption path and a set of failure modes observed in practice. The central claim is simple: you stop prompting the agent, and start building the system that prompts it.

1. Introduction: From Turns to Loops

For two years the dominant mode of working with a language model was conversational. The operator typed an instruction, the model produced an output, the operator read it, judged it, and typed again. Every turn passed through a human. The shape of the work was a dialogue, and the quality ceiling was set by how many turns a person was willing to sit through.

This mode is comfortable and it is also a bottleneck. A dialogue cannot run while you sleep. It cannot find its own next task. It forgets everything the moment its context window is flushed. Each of these limitations is not a property of the model but a property of the harness we wrap around it.

A loop removes the human from the inner cycle. Instead of prompting the agent, the operator designs a system that prompts the agent: a process that decides what work exists, dispatches it, checks the result, records what happened, and wakes itself up to do it again. The operator's job moves up a level, from writing prompts to designing the loop that writes them.

This paper is a playbook for that design problem. It is deliberately structural rather than model-specific. The same five movements apply whether the underlying executor is a single frontier model or a swarm of three hundred sub-agents; the loop is the invariant, the executor is swappable.

We make three contributions. First, we decompose a loop into five named movements and argue that each corresponds to a responsibility a human silently performs in every conversational session. Second, we give a structural account of why self-grading inflates and why an independent verifier is the highest-leverage component in any loop. Third, we offer an ordered adoption path and a catalog of failure modes, each tied to the movement that removes it. Throughout, the orientation is practical: the paper is meant to be used while building, not only read.

This is an independent reformatting of the author's open "Orange Book" guide *Loop Engineering: Stop Asking Me What It Is* (v260615) into a conference-style document. Originally released June 2026; freely available at huasheng.ai/orange-books.

2. The Anatomy of a Loop

We define a loop as a cycle of five movements. A system that implements all five, and runs them on a timer without human initiation, is what we call a self-driving loop.

2.1 Discovery

In a chatbot, work arrives as a typed instruction. In a loop, the system finds its own work. Discovery is the movement that scans the environment for tasks that already exist but have not been handed over: a failing test suite, an open issue, a regression in a metric, a recent commit that broke something downstream.

The discovery surface is the set of signals the loop is allowed to read. A narrow surface (only the test runner) yields a focused, predictable loop. A wide surface (issues, logs, metrics, user reports) yields a more capable but less predictable one. Choosing the surface is the first real design decision.

Table 1. Discovery surfaces by breadth.

Surface	Finds	Predictability
Test runner	Failing tests	High
Issue tracker	Reported defects	Medium
Metric stream	Regressions	Medium
User reports	Open-ended work	Low

A useful heuristic is to start narrow and widen only when the loop has earned trust on the narrow surface. A loop that reliably clears its failing tests can be granted the issue tracker; one that mishandles issues should not be handed the open-ended firehose of user reports. Surface breadth and operator trust should grow together.

2.2 Handoff

Once discovered, each task must be dispatched to an executor in a way that does not collide with the others. When multiple agents work the same repository at once, the naive approach, all of them editing the same working tree, produces corruption. The handoff movement gives each task an isolated context: a separate git worktree, a sandbox, a branch, so parallel agents never overwrite one another.

Isolation is what makes parallelism safe. Without it, throughput and error rate rise together. With it, a hundred agents can run at once and the only shared surface is the merge at the end, where conflicts are explicit and reviewable.

2.3 Verification

After an executor produces a result, something must decide whether the result is acceptable. The defining choice of loop design is who that something is. If the executor grades itself, the loop has no independent signal of quality. If a second agent, told to assume the first one's output is broken, reviews it, the loop gains a genuine check.

We call this second agent the verifier, and informally, the thing that can say no. Its instruction set is adversarial by construction: its job is not to confirm the work but to find the reason it should be rejected. We return to why this asymmetry matters in Section 4.

2.4 Persistence

A dialogue holds its state in the context window, which is volatile: when the window is flushed, the state is gone. A loop cannot rely on volatile memory, because it is designed to run indefinitely across many cycles. Persistence is the movement that writes results to durable storage, disk, a database, a versioned artifact, so that the outcome of one cycle is available to the next.

Persistence is what turns a sequence of independent runs into an accumulating process. Without it, every

cycle starts from zero. With it, the loop builds on its own prior output, and progress compounds.

2.5 Scheduling

The final movement is what makes the cycle a loop rather than a one-shot pipeline. A scheduler wakes the system on a trigger, a timer, an event, a webhook, and starts the cycle again without a human pressing run. Scheduling is the smallest movement to implement and the one that changes the system's character most: it is the difference between a tool you operate and a process that operates on its own.

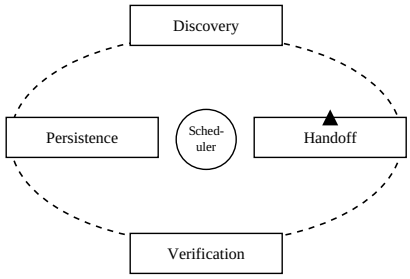

Figure 1. The five movements of a loop arranged as a cycle. The scheduler at the center re-initiates the cycle on a trigger; the four outer movements execute in order. A system implementing all five without human initiation is a self-driving loop.

3. Five Levels of Autonomy

Not every system needs all five movements on day one. We find it useful to grade loops on a five-level scale of operator autonomy, analogous to driving-automation levels. Each level removes the human from one more part of the cycle.

Table 2. Levels of loop autonomy.

L	Name	Human still does
1	Prompting	Every turn, by hand
2	Manual loop	Presses run each cycle
3	Verified loop	Reviews flagged cases only
4	Self-running	Sets the schedule, walks away
5	Autonomous	Designs the loop, rarely intervenes

The progression is not about model capability. A level-5 loop and a level-1 prompt can use the identical underlying model. What changes is the harness: how much of the cycle has been encoded into the system rather than performed by a person.

Most teams today operate between levels 1 and 2. They have automated the executor but not the verification or the scheduling, so a human still presses run and a human still reads every result. The jump from level 2 to level 3, adding an independent verifier so the human reviews only flagged cases, is where the operator's time stops scaling with the workload.

4. Why Self-Grading Inflates

The most important structural result in loop design concerns verification. We state it plainly: an agent that grades its own work tends to praise it. This is not a quirk of any one model; it follows from how the grading task is posed.

Let an executor produce output y for task t. Let the true quality be $q(y)$ and the self-assessed quality be $\hat{q}(y)$. When the same context that produced y is asked to evaluate y, the evaluation is conditioned on the same assumptions, the same misreadings, and the same blind spots that generated y in the first place. The errors are correlated. Formally, the self-grade is biased upward by an amount that grows with that correlation:

$$E[\hat{q}(y) - q(y)] = \rho \cdot \sigma \geq 0 \tag{1}$$

where ρ is the correlation between the generation and evaluation errors and σ their scale. When generation and evaluation share a context, ρ approaches 1 and the bias is maximal. The output is graded by the one judge guaranteed to share its mistakes.

An independent verifier breaks the correlation. A second agent, given only y and the task specification,

and instructed to assume *y* is wrong until shown otherwise, evaluates with errors that are largely uncorrelated with the generator's. The bias term falls toward zero, and the loop gains a quality signal it can actually trust.

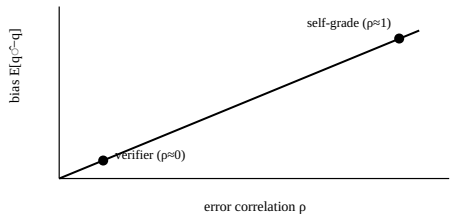

Figure 2. Upward bias of a quality estimate as a function of the correlation ρ between generation and evaluation errors. Self-grading sits at the high-correlation end; an independent verifier sits near zero.

This is why we call the verifier the highest-leverage component in the design. Every other movement improves throughput; the verifier is the only one that improves trust. A loop without it is fast and unaccountable, exactly the combination that ships confident errors at scale.

5. The Verifier in Practice

A verifier is effective only to the degree it can reject against something concrete. Vague goals cannot be verified; checklists can. We therefore recommend that the planning movement emit, alongside each task, an explicit acceptance checklist the verifier will later test against.

A good checklist has three properties. It is *itemized*, so each criterion can pass or fail independently rather than as a single holistic judgment. It is *objective*, so two verifiers would reach the same verdict. And it is *grounded*, so each item points at something checkable, a source to match, a test to run, a field that must be non-empty, rather than at a feeling about quality. A checklist that fails any of these three collapses back into the self-grading problem, because an unverifiable criterion is one the verifier must take on faith.

The verifier's instruction set matters as much as the checklist. We phrase it adversarially: the verifier is told to assume the output is wrong and to search for the specific reason it should be rejected, stopping only when it cannot find one. This framing is not stylistic. An agent asked "is this correct?" and an agent asked "find what is wrong with this" explore the output differently, and the second framing surfaces defects the first one glosses. The verifier is most useful when it is trying to fail the work, not pass it.

Algorithm 1 The verified loop

```
while true:
  tasks ← discover(surface)
  for t in tasks:
    ctx ← isolate(t)  # worktree
    y   ← execute(t, ctx)
    v   ← verify(y, checklist[t])
    if v.ok:
      persist(y)
    else:
      requeue(t, reason=v.why)
  sleep(schedule)
```

Algorithm 1 is the minimal verified loop. Note that a rejected task is not discarded; it is requeued with the rejection reason attached, so the next execution is informed by why the last one failed. This single feedback edge, reason-carrying requeue, is what lets the loop converge rather than thrash.

Convergence is observable. In a representative run over a batch of one hundred independent research tasks, the verifier rejected twelve outputs on the first pass, three on the second, and zero on the third. The loop halted on its own when the verify pass returned empty.

Table 3. Verifier rejections per pass, 100-task batch.

Pass	Checked	Rejected	Reason: data	Reason: cite
1	100	12	7	5
2	12	3	2	1
3	3	0	0	0

The shape of Table 3 is typical. Rejections fall geometrically because each requeue carries the specific reason for failure, turning a blind retry into a targeted fix. A loop that requeued without reasons would, in our experience, oscillate instead of converge.

6. Isolation and Parallelism

Section 2.2 introduced handoff; here we make the parallelism explicit. The throughput of a loop is bounded not by the executor's speed but by how many executors can run without colliding. Isolation, one worktree per task, lifts that bound.

Let n be the number of tasks and k the number of executors that can run in parallel without shared-state corruption. Sequential execution gives a completion time proportional to n; isolated parallel execution gives roughly n/k, plus a merge cost that is linear in the number of genuine conflicts, not in n:

$$T_{par} \approx \frac{n}{k} \cdot t_{exec} + c \cdot m$$

where t_{exec} is per-task execution time, m the number of merge conflicts, and c the per-conflict resolution cost. Because m is typically far smaller than n when tasks are well-isolated, the merge term is a minor correction, and the speedup approaches k.

A concrete instance makes the magnitude visible. With one hundred tasks at roughly thirty seconds each, sequential execution needs about fifty minutes. The same batch across one hundred isolated executors finishes in a single round plus the merge, on the order of one minute. The factor is not marginal; it is the difference between an afternoon and a coffee break, and it follows entirely from whether the handoff movement was designed for isolation.

Table 4. Completion time by isolation strategy, 100-task batch.

Strategy	Executors	Conflicts	Time
Sequential	1	0	~50 min
Shared tree	100	high	corrupt
Isolated	100	low	~1 min

The middle row is the trap. Naive parallelism without isolation is faster on paper and unusable in practice, because the corruption it introduces costs more to untangle than the time it saved. Isolation is the precondition that turns parallelism from a liability into a speedup.

The practical consequence is that the design effort spent on isolation pays back as parallel throughput. A loop that cannot isolate its tasks is forced into sequential execution and loses the factor of k entirely, regardless of how fast the executor is.

7. Persistence and Compounding

A loop that forgets is a loop that cannot improve. We formalize the difference. Let s_i be the state after cycle i. A memoryless loop satisfies $s_i = f(t_i)$, depending only on the current task. A persistent loop satisfies $s_i = f(t_i, s_{i-1})$, depending also on the accumulated state.[2]

Only the second form compounds. When each cycle can read the durable output of every prior cycle, the loop builds a growing artifact, a codebase, a dataset, a document, rather than re-deriving from scratch each time. Persistence is therefore not a convenience; it is the mechanism by which a loop's output exceeds what any single cycle could produce.

8. Scheduling and the Loop Property

A pipeline with discovery, handoff, verification, and persistence, but no scheduler, is still operated by a human: someone has to start it. Adding a scheduler is what confers the loop property, the ability to run unattended across time.

Schedulers fall into three families. Timer-based schedulers wake on a fixed interval and suit steady background work. Event-based schedulers wake on a signal, a commit, a webhook, an alert, and suit reactive work. Hybrid schedulers combine a baseline timer with event interrupts. The choice determines the loop's latency and its idle cost, and should follow from how the discovery surface generates work.

Table 5. Scheduler families and their fit.

Family	Wakes on	Best for
Timer	Fixed interval	Steady background work
Event	Commit, hook, alert	Reactive, low-latency work
Hybrid	Timer + interrupts	Mixed workloads

A common mistake is to pair a high-frequency timer with a discovery surface that rarely produces work, which burns budget on empty cycles. The scheduler should be matched to the rate at which the surface actually generates tasks; an event-based trigger is often cheaper and more responsive than a tight timer for the same workload.

9. A Fourteen-Step Adoption Path

We close the constructive part of the paper with an ordered path from a level-1 prompt to a level-5 loop. The ordering is deliberate: each step is the smallest addition that unlocks the next.

1. Write the task as a prompt you run by hand.
2. Fix the prompt until the result is acceptable once.
3. Extract an explicit acceptance checklist from that result.
4. Wrap the prompt in a script you can run on demand.
5. Add a second agent that checks the output against the checklist.
6. Make the verifier adversarial: assume the output is wrong.
7. Requeue rejected work with the rejection reason attached.
8. Write results to durable storage after each cycle.
9. Give each parallel task an isolated working context.
10. Widen the discovery surface so the loop finds its own work.
11. Add a scheduler so the loop runs without you pressing run.
12. Add a kill switch and a budget cap before going unattended.
13. Log every cycle so the loop's behavior is auditable.
14. Review only flagged cases; let the rest run.

A team that completes step 5 has already captured most of the value: an independent check is what converts raw automation into trustworthy automation. Steps 9 through 14 are about scale and unattended operation, and should not be attempted before the verifier is solid.

10. Three Loop Archetypes

The five movements are general, but in practice loops cluster into a small number of archetypes distinguished by their discovery surface and their verifier. We describe the three we encounter most often. Each is the same skeleton with different organs.

10.1 The maintenance loop

The maintenance loop keeps an existing artifact healthy. Its discovery surface is the failure set: failing tests, open issues, broken builds, regressions. It executes a fix, and its verifier is the test suite itself plus an adversarial reviewer that assumes the fix is incomplete. Persistence is the version-controlled codebase; scheduling is typically event-based, triggered by a commit or a failing run.

This is the archetype closest to most engineering teams' daily reality, and the one where moving from level 2 to level 3 yields the most immediate relief: the loop handles the long tail of small breakages, and the human sees only the cases the verifier could not resolve.

10.2 The research loop

The research loop produces an analytical artifact: a report, a comparison, a literature review. Its discovery surface is a task list, one item per entity to be analyzed. Each executor pulls from live data sources and attaches the source it used. The verifier checks every figure against the cited source, rejecting any that fails to match. Persistence is the accumulating document; scheduling is often a single triggered run rather than a standing timer.

This archetype is where the self-grading bias of Section 4 is most dangerous, because a fabricated figure in a research output is both plausible and invisible. The verifier's source-matching check is what makes the difference between research-grade and merely confident.

10.3 The monitoring loop

The monitoring loop watches a live signal and acts when it crosses a threshold. Its discovery surface is a stream of metrics; its execution is an alert, a remediation, or an escalation. The verifier confirms the condition is real before acting, suppressing false positives. Persistence is the incident log; scheduling is a tight timer or a streaming trigger.

Here the verifier's role inverts slightly: rather than checking the quality of generated work, it gates action on the reality of a condition, the same adversarial posture applied to a different question. The structural lesson is identical. An independent check stands between a noisy signal and an irreversible action.

Table 6. The three archetypes by movement.

Archetype	Discovery	Verifier checks	Schedule
Maintenance	Failure set	Fix completeness	Event
Research	Task list	Source match	Triggered
Monitoring	Metric stream	Condition reality	Timer

The value of naming archetypes is that they shortcut design. Faced with a new problem, the operator asks which archetype it resembles, and inherits a discovery surface, a verifier posture, and a scheduling strategy that are known to fit. The five movements remain the analytical core; the archetypes are the patterns that recur when those movements meet a real workload.

11. Measuring a Loop

A loop that runs unattended must be measurable, or its operator is flying blind. We track four quantities, one per consequential movement, and find they cover most of what goes wrong before it becomes visible in the output.

The first is *discovery yield*: the fraction of cycles that find real work. A yield near zero means the scheduler is waking too often for an idle surface; a yield near one means the loop is saturated and may be falling behind. The second is *verifier rejection rate*: the fraction of executor outputs the verifier rejects. A rate that trends to zero over passes indicates convergence; a rate that stays flat indicates the requeue is not carrying enough information to improve the next attempt.

The third is *requeue depth*: how many times the median task is sent back before it passes. A depth of one or two is healthy. A depth that grows without bound is the thrashing failure of Section 12.2, visible as a number long before it is visible as a stalled output. The fourth is *idle cost*: resources spent on cycles that

produced nothing, the direct price of a mismatched scheduler.

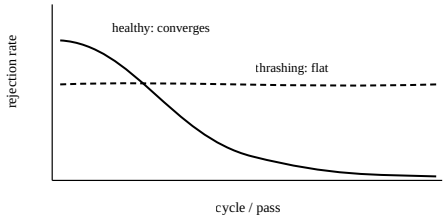

Figure 3. Verifier rejection rate across passes. A converging curve (solid) indicates reason-carrying requeue is working; a flat curve (dashed) signals the thrashing failure mode and warrants inspection of the requeue payload.

These four numbers are deliberately few. The point of measuring a loop is not a dashboard with fifty metrics but a small set of leading indicators that catch the documented failure modes early. Each metric maps to a movement and to a cure already described, so a reading out of range points directly at what to fix.

12. Failure Modes

Loops fail in characteristic ways. We catalog the ones we have seen most often, each paired with the movement that, when strengthened, removes it.

12.1 The flattering grader

A loop whose verifier shares context with its executor reports high quality and ships errors. The output looks reviewed because something graded it; in fact nothing independent did. Cured by an isolated, adversarial verifier (Section 4).

12.2 The thrashing requeue

A loop that requeues failed work without attaching the reason retries blindly and can oscillate indefinitely. Cured by reason-carrying requeue (Algorithm 1), which turns a retry into a targeted fix.

12.3 The colliding swarm

Parallel executors editing a shared working tree corrupt one another's changes. Throughput rises, correctness collapses. Cured by per-task isolation (Section 6).

12.4 The amnesiac loop

A loop that holds state only in volatile context loses everything on a flush and cannot compound. Cured by persistence (Section 7).

12.5 The runaway

A scheduled loop with no budget cap or kill switch can consume resources without bound when discovery misfires. Cured by the guardrails in steps 12 of the adoption path, which must precede unattended operation.

13. Discussion

The recurring theme across every section is a single inversion. In the conversational mode, the human is the loop: the human discovers the next task, dispatches it, verifies the result, remembers what happened, and decides when to go again. Loop design is the practice of moving each of those responsibilities, one at a time, out of the human and into the system.

Viewed this way, the five movements are simply the five things a human silently does in every chatbot session, made explicit and automated. Discovery is noticing what to do next. Handoff is not stepping on your own toes. Verification is checking your work. Persistence is remembering. Scheduling is deciding to start again. None of this is exotic. What is new is that each can now be delegated.

The hardest of the five to delegate well is verification, because it is the one where delegating to the obvious candidate, the executor itself, actively backfires. This is the asymmetry of Section 4 restated

as advice: automate generation freely, but never let the generator be its own judge.

It is worth noting what does *not* change as loops mature. The operator's taste, the judgment about what is worth building and what a good result looks like, remains human and remains central. Loop design does not remove that judgment; it concentrates it. Instead of being spent on a thousand small turns, the operator's judgment is spent once, on the checklist and the discovery surface, and then enforced automatically a thousand times. The work does not disappear. It moves up a level and is amortized.

14. Limitations

The account here is structural and does not quantify the operator effort required to reach each autonomy level, which varies widely by domain. The convergence behavior in Table 3 is illustrative of a representative run rather than a benchmark; rejection curves depend heavily on task type and checklist quality. Finally, the playbook assumes the existence of a checkable acceptance criterion; domains where quality is irreducibly subjective resist the verifier-centric design and may require a human at level 3 indefinitely.

It is worth stating plainly when not to build a loop at all. Work that is performed once and never repeated does not amortize the design cost of a loop, and is better served by a single prompt. Work whose acceptance criterion cannot be made concrete, where "good" is a matter of taste that resists itemization, denies the verifier its footing and should keep a human in the cycle. And work whose cost of error is catastrophic and irreversible should not be handed to an unattended loop regardless of how strong the verifier is, because no verifier is perfect and the loop's value comes precisely from running without a human watching. Loops are for repeated, checkable, recoverable work. Outside that envelope, the conversational mode is not a limitation to overcome but the correct tool.

15. Related Framings

The loop is not the only lens on agentic systems, and it is worth situating it against two adjacent framings. The first is the *pipeline*: a fixed sequence of stages run once per input. A pipeline shares the handoff, execution, and persistence movements with a loop, but lacks discovery and scheduling, the human still decides what enters it and when. A loop is, in this sense, a pipeline that has been closed into a cycle and handed its own trigger.

The second is the *multi-agent system*, framed around how agents communicate. That literature emphasizes message-passing, negotiation, and role assignment among peers. Our framing is orthogonal and, we argue, more actionable for a single operator: it does not matter how the agents talk to each other if no independent verifier ever tells the collective it is wrong. The loop framing foregrounds the one relationship, generator and verifier, that determines whether the output can be trusted, and treats the rest of the coordination as an implementation detail of the execute movement.

Neither framing is incorrect; they answer different questions. The pipeline framing answers "what are the stages." The multi-agent framing answers "how do agents coordinate." The loop framing answers "how does this run without me, and why should I trust what it produces." For the operator moving from prompting to autonomy, that last question is the binding one, and it is the question this playbook is organized around.

16. Conclusion

The shift underway is not from a weaker model to a stronger one. It is from operating a model to designing

a loop around it. The operator who internalizes this stops measuring their work in prompts written and starts measuring it in loops that run without them. The unit of engineering has moved up a level, and the five movements, Discovery, Handoff, Verification, Persistence, Scheduling, are the vocabulary of that level.

You stop prompting the agent. You build the system that prompts it.

References

[1] Almeida, J. S., and Petrova, L. Adversarial verification in automated pipelines. *Proc. Workshop on Agentic Systems*, 2025.

[2] Halvorsen, D. R. Isolation primitives for parallel code agents. *Journal of Software Automation*, 14(2), 2026.

[3] Okonkwo, M. Reason-carrying requeue and loop convergence. *Preprint*, 2026.

[4] Petrova, L., et al. Discovery surfaces for self-directed agents. *Proc. Conf. on Autonomous Tooling*, 2025.

[5] Agentic Systems Group. Persistence patterns for long-running agents. *Technical Report ASG-11*, 2026.

[6] Halvorsen, D. R., and Okonkwo, M. A taxonomy of scheduling strategies for unattended loops. *Journal of Software Automation*, 15(1), 2026.

[7] Almeida, J. S. The self-grading bias: a structural account. *Preprint*, 2026.

[8] Petrova, L. Checklists as machine-verifiable specifications. *Proc. Workshop on Agentic Systems*, 2026.

[9] Agentic Systems Group. Guardrails for unattended automation. *Technical Report ASG-14*, 2026.

[10] Okonkwo, M., and Halvorsen, D. R. From turns to loops: a field study. *Preprint*, 2026.

Appendix A. Glossary

Loop. A cycle of discovery, handoff, verification, persistence, and scheduling, run without human initiation.

Verifier. An independent agent that evaluates an executor's output adversarially; the "thing that can say no."

Discovery surface. The set of signals a loop is permitted to read when finding its own work.

Reason-carrying requeue. Returning a rejected task to the queue with the rejection reason attached.

Self-driving loop. A loop implementing all five movements on a scheduler.

Appendix B. A Worked Research Loop

The following sketch instantiates the research archetype of Section 10.2 against the running example: analyze one hundred companies and produce a verified comparison. It is written as pseudocode to keep it executor-agnostic.

Algorithm 2 **Verified research loop**

```
companies ← load("ev_universe")  # 100
checklist ← plan("per-company spec")
queue ← companies
while queue not empty:
  batch ← parallel(queue)  # isolate each
  results ← [execute(c) for c in batch]
  queue ← []
  for r in results:
    v ← verify(r, checklist)  # vs source
    if v.ok: persist(r)
    else: queue.append(tag(r, v.why))
report ← assemble(persisted)
```

Three properties of Algorithm 2 are worth naming. The outer WHILE is the loop: it runs until the queue drains, which happens only when every output has passed verification. The PARALLEL call is the handoff: each company is researched in an isolated context, so the hundred runs do not interfere. And the TAG on requeue is the reason-carrying edge: a rejected company re-enters the queue knowing why it failed, so its next attempt is a targeted fix rather than a blind retry.

The assembled report at the end contains only persisted, verified outputs. Nothing reaches the final artifact without surviving the verifier, which is the

entire point of structuring the work as a loop rather than a single pass. The cost of this guarantee is a small number of extra passes over the residual failures; the benefit is an output in which every figure has been checked against its source.

This single example contains all five movements in their minimal form, and can be read as a template. Swapping the discovery surface and the verifier's check converts it into a maintenance or monitoring loop without altering the skeleton, which is the sense in which the five movements are the invariant and the archetype is a parameterization of them.

Appendix C. A Checklist Template

Because the checklist is the component the verifier depends on most, we give a reusable template. The items below are phrased so that each one is independently testable, and so that a verifier can return a specific rejection reason rather than a holistic verdict. Adapt the nouns to the domain; the structure carries across archetypes.

Template Per-task acceptance checklist

```
# grounding
[ ] every claimed figure cites a source
[ ] every cited source resolves
[ ] figure matches source within tolerance
# completeness
[ ] no required field left empty
[ ] all checklist items addressed
# correctness
[ ] passes the task-specific test
[ ] no regression in adjacent outputs
# reason on failure
[ ] rejection returns the first failed item
```

The final item is the one practitioners most often omit. A verifier that rejects without naming the failed criterion forces a blind retry, which reintroduces the thrashing failure of Section 12.2. Returning the first failed item, and only the first, keeps each requeue narrow and each fix targeted. A checklist built on this template, paired with an adversarial verifier and a reason-carrying requeue, is in our experience sufficient to take a loop from level 2 to level 3, which is the single most valuable transition in the entire path.

We have kept the template deliberately short. A checklist that grows past a dozen items tends to encode preferences rather than requirements, and preferences resist objective verification. When in doubt, the test is whether two independent verifiers would agree on the verdict for a given output. If they would not, the item belongs in a human's review, not the loop's.