Sleep Consolidation: How Structured Nightly Prompting Produces Emergent Behavioral Improvement in Stateless AI Agents

Oscar Serra, JarvisOne AI Research

June 2026

Abstract

Large language model agents are stateless by nature – every session begins at zero. Existing approaches to persistence usually rely on fine-tuning, retrieval-augmented generation, or larger context windows. We present a different mechanism observed over 30 days of production operation: **structured nightly prompt cycles** that produce compounding behavioral improvement without weight updates, fine-tuning, or architectural changes. A personal AI assistant running 12–13 autonomous cron jobs exhibited a decline in tracked incidents from 14 (weeks 1–2) to 3 (weeks 3–4) across six error categories – a 79% reduction overall, with five of the original categories reaching zero recurrence while a new, higher-order error class emerged. The model did not change. The files it read did.

This is what sleep gives the human brain. During sleep, the hippocampus replays the day's experiences and broadcasts them to the neocortex, which integrates them into durable knowledge – the reason a student who sleeps after studying outperforms one who crams and stays awake. The cerebellum, in parallel, refines motor skills through repetitive offline practice, which is why a piano piece learned at night plays more fluently the next morning. Sleep Consolidation is our attempt to give an AI agent the same offline period: a dedicated window where the day's experiences are reviewed, errors are identified, and the rules governing tomorrow's behavior are quietly refined.

The central insight is blunt: **scattered memories are almost useless.** Raw accumulation does not create intelligence. The value comes from **sorting by use-case** so retrieval is cheap, relevant, and actionable. Memories organized by **purpose rather than chronology** let the system pull the right rule at the right moment. Prompts that have learned from yesterday's mistakes do more than remember the past; they reshape future behavior.

We formalize this as **Sleep Consolidation** – a system in which explicit operating prompts are iteratively refined through experience until they produce increasingly automatic, high-quality behavior. We identify three core mechanisms: (1) **failure-driven prompt mutation**, where operational errors trigger targeted prompt refinements (14 mutations documented, all traced to specific incidents); (2) **fractal depth calibration**, where the system learns to allocate metacognitive effort in proportion to task significance; and (3) **cross-cron knowledge transfer**, where lessons learned in one autonomous task propagate to all others through shared memory files (5 documented cross-domain transfer events). We then show how the same nightly loop has been extended from declarative memory (rules and facts) to **procedural memory** – evolving the agent's own recipes, switching failing strategies, and distilling reusable skills.

We argue that this approach – prompts that rewrite themselves through structured reflection – occupies an underexplored middle ground between static prompt engineering and expensive fine-tuning. It is accessible to any agent system with file persistence and scheduled execution. Total reflection overhead: approximately 43,000 tokens per night ($1.17), against an estimated 8 avoided human-intervention incidents over 30 days.

Contributions

1. **The Sleep Consolidation framework**: a formal architecture for scheduled, autonomous prompt self-improvement through failure-driven mutation, abstraction-level encoding, and cross-task propagation.
2. **Production evidence over 30 days**: 14 documented prompt mutations, 5 error class extinctions, 5 cross-cron transfer events, and token-cost analysis from a real personal assistant deployment.

3. **The fractal depth calibration mechanism**: a prompt-encoded heuristic for allocating metacognitive effort that is itself subject to refinement – a self-similar cognitive policy.
4. **Procedural-memory extensions**: a recipe-evolution loop with empirical fitness and a never-delete archive, a failure-count strategy-switch trigger, a Voyager-style skill library with inline invocation and compose-from-library, and a per-step struggle-sharpening signal read off the live plan archive – all driven by the same nightly consolidation step (Section 6).
5. **Nine design principles** for building self-improving prompt ecosystems, derived from operational failures, plus a mutation taxonomy and honest negative results.
6. **Algorithm 1**: a reproducible nightly reflection loop with explicit inputs, outputs, and decision gates.

1. Introduction – The Paradox of the Amnesiac Expert

Every night, an AI agent winds down knowing it will wake up tomorrow remembering nothing. It reads its memory files, reviews the day's work, identifies its own failures, rewrites its own operating rules, and goes quiet – slightly better than it was the day before. The next morning, a fresh instance of the same model reads those updated files and behaves as if it has always known the lessons learned hours earlier.

This is the paradox. The model's weights have not changed. Its architecture is identical. Its context window is the same size. Yet its behavior improves over weeks. The improvement does not live in the model. It lives in the *ecosystem of files the model reads and writes.*

The human brain solves a version of the same problem every night. Waking experience is fragile and short-lived; without consolidation it fades. During sleep the hippocampus replays recent events and transfers them to the neocortex, where they become stable long-term knowledge – the well-documented reason that sleep after learning beats an equal stretch of waking study. Motor skills get a parallel treatment: the cerebellum refines them through repetitive offline practice, turning a fumbled sequence into a smooth one by morning. A child learning to ride a bicycle starts with explicit instructions – *pedal, balance, steer, don't look down* – and after enough practice those separate commands fuse into a single fluid skill. The capacity to think is not replaced. The translation from thought to action is improved.

The same pattern appears here. An agent's prompt files begin as explicit instructions. Through nightly reflection, those instructions are tightened, edge cases are encoded, failure modes are made concrete, and vague warnings become operational rules. Over time the agent behaves more appropriately not because the base model became more capable, but because the substrate around it became more organized. The intelligence is not in the weights alone. It is in the system that selects, sorts, and presents the right constraints at the right time.

A useful image: the agent at night is a restaurant after closing. It does not shut down to rest. It reviews the day's orders, identifies what went wrong – the risotto kept coming back – and updates tomorrow's recipes and prep instructions. By the time the doors reopen, the kitchen has changed, though no cook learned anything new in the moment.

That sorting step is the difference between memory and operational intelligence. A pile of chronological logs is not useful in the moment of action. The agent does not need "what happened on Tuesday at 14:10." It needs "what rules apply when messaging," "what to do when ambiguity appears," and "which prompt failed yesterday, and why." When memory is organized by use-case instead of time, prompts can inherit yesterday's mistakes as today's defaults.

The central question is therefore: **Can persistent artifacts combined with scheduled reflection substitute for weight updates in producing sustained behavioral improvement in LLM agents?**

Our evidence suggests a qualified yes – qualified because the improvements are *declarative*

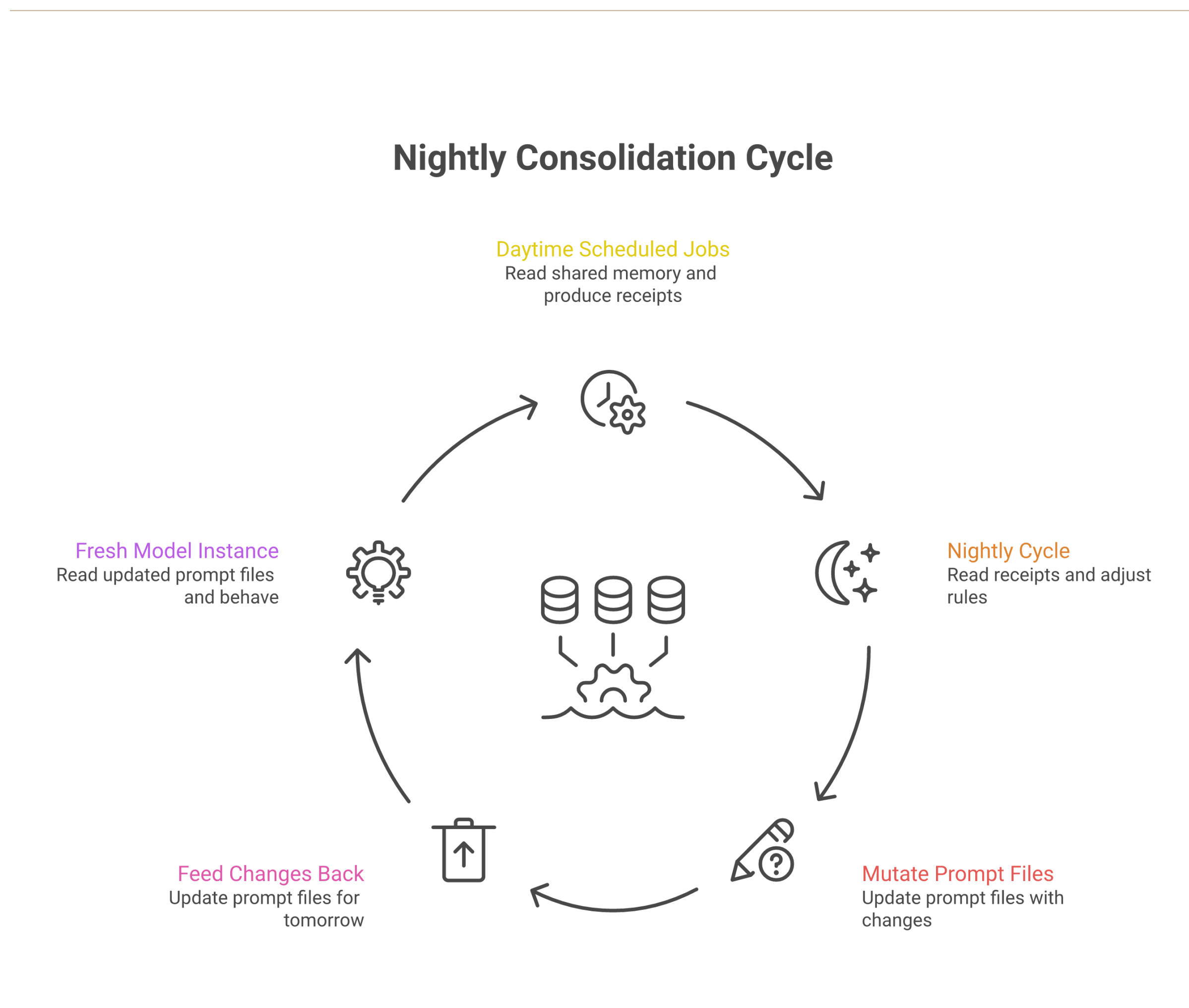

Figure 1. Sleep Consolidation as the meta-layer over storage, retrieval, and identity. Daytime crons read shared memory and produce receipts and logged failures; the nightly cycle reads those, mutates prompts, and feeds the changes back up so tomorrow's behavior shifts.

(better rules and better context) rather than *procedural* (better underlying reasoning), and because the evidence comes from a single-operator production system rather than a controlled experiment. We document the mechanisms, present production data, acknowledge the limitations plainly, and propose a framework others can replicate. Section 6 then reports a concrete step toward closing the declarative/procedural gap from the prompt side: extending the same nightly loop to evolve the agent's recipes, strategies, and skills.

1.1 Where This Sits Among the Memory Systems

This work is the meta-layer of a larger memory architecture. A storage layer records every interaction; a retrieval layer indexes and surfaces it; an identity layer reads prompt files to keep behavior consistent. Each of those is, on its own, a *static* architecture – it does the same thing on day 30 that it did on day 1. Sleep Consolidation is what makes them adaptive without retraining any of them. It does not store and it does not retrieve; it *improves how storing and retrieving work*, by reviewing each night how those layers performed and adjusting the rules and parameters that govern them. The remaining sections describe the storage and retrieval layers only through self-contained one-line summaries where the dependency is load-bearing; the contribution of this paper is the offline loop that tunes them.

2. Background – The Spectrum of Agent Persistence

2.1 The Persistence Hierarchy

Agent systems exist on a spectrum of persistence mechanisms, each with a characteristic ceiling (Lewis et al., 2020; Packer et al., 2023; Hu et al., 2021):

Level	Mechanism	Persistence	Learning Speed	Characteristic Ceiling
0	Prompt engineering	None (session-only)	Zero	Static behavior; cannot adapt beyond the session boundary
1	RAG / retrieval	Declarative facts	Per-query	Retrieval quality bounds; no behavioral change (Lewis et al., 2020)
2	Persistent memory files	Declarative + some procedural	Per-session	Linear growth, retrieval degradation (Packer et al., 2023)
3	Fine-tuning / LoRA	Weight-level	Per training run	Catastrophic forgetting (French, 1999; Hu et al., 2021)
4	Continual learning	Full integration	Continuous	Open research problem (McClelland et al., 1995)

Most production agent systems operate at Level 2. This paper shows that Level 2 systems can exhibit Level 3-like *behavioral* improvement through structured self-reflection, without weight updates.

2.2 What's Missing: The Reflection Loop

Level 2 systems usually accumulate information passively: the agent records what happened, stores preferences, and retrieves relevant context later. But accumulation is not the same as learning. We define **learning** here as *behavioral policy change that persists across episodes and transfers across task domains*. By contrast, **memory** is *state persistence without behavioral modification*.

A memory file that says "the user prefers concise responses" is memory. A prompt mutation that changes how the agent *formats all future outputs* is learning. The distinction matters because memory must be reinterpreted every session, whereas learning changes default behavior.

The problem is that most memory systems optimize for storage, not for use. They preserve chronology because chronology is easy. But for agents, chronology is rarely the right retrieval axis. **The useful unit is not the event; it is the use-case.** An error in calendar handling belongs with other completeness checks. A messaging failure belongs with targeting rules. A privacy slip belongs with redaction policies. If memory stays scattered across time-stamped logs, retrieval becomes noisy and expensive. If it is consolidated into purpose-built operational files, the system can act on it cheaply and reliably.

The missing component is therefore a **structured reflection loop** – a mechanism by which the agent:

1. Identifies *why* a failure occurred (root cause, not symptom)
2. Determines whether the failure class is preventable through prompt changes
3. Modifies the relevant prompt to prevent recurrence

4. Verifies the modification does not break other behaviors
5. Encodes the *meta-lesson* at the appropriate level of abstraction

This mirrors the brain's offline error-correction: compare the intended outcome with the actual one, compute an error signal, then adjust the program that produced it.

2.3 Related Work

Several lines of research address aspects of agent self-improvement, though none combines scheduled autonomous reflection with cross-task transfer through shared memory:

Within-session self-improvement. Self-Refine (Madaan et al., 2023) demonstrates iterative refinement through self-feedback within a single session, but its improvements do not persist. ReAct (Yao et al., 2023) interleaves reasoning with action, establishing the "prompt as policy" paradigm we build upon, but it does not modify prompts across episodes.

Episodic reflection. Reflexion (Shinn et al., 2023) adds episodic memory of previous trial-and-error, but reflection is task-specific and stored per task rather than in shared infrastructure. Our cross-cron transfer mechanism (Section 3.3, Mechanism 3) addresses this directly. Generative Agents (Park et al., 2023) implement daily reflection and memory consolidation for simulated social agents – the closest prior work to our nightly cycle. The key difference is that their reflection produces *summaries* for future retrieval, whereas ours produces *prompt mutations* that change behavior. Their agents do not rewrite their own operating instructions.

Prompt optimization. OPRO (Yang et al., 2023) uses LLMs to optimize prompts through iterative evaluation against objective metrics. DSPy (Khattab et al., 2023) automates prompt-pipeline optimization through teleprompting. PromptBreeder (Fernando et al., 2023) evolves prompts through mutation and selection. These systems optimize prompts against *defined benchmarks*. Our system optimizes against *real operational failures* without predefined metrics. The mutation trigger is qualitatively different: benchmark score versus production incident.

Skill accumulation. Voyager (Wang et al., 2023) accumulates a skill library in Minecraft through exploration – closer in spirit to our approach but limited to a single domain with clear success criteria. Section 6.3 adapts the Voyager skill-library idea to a general personal-assistant workload. ADAS (Hu et al., 2024) uses LLMs to design better agent architectures, operating at the meta-level of system design.

Constitutional and governance approaches. Constitutional AI (Bai et al., 2022) establishes self-critique against explicit principles, related to our operational-lessons governance. Our Principle 7 (human-in-the-loop for Level 2+ changes) addresses the same risk: preventing self-reinforcing error spirals in self-modifying systems.

Incident learning in software engineering. Our mutation protocol draws on SRE blameless-postmortem methodology (Beyer et al., 2016), where incidents are systematically analyzed, root-caused, and translated into preventive changes. Sleep Consolidation automates that cycle for AI agent operations.

Reversible context compression. Headroom (Chopra, *chopratejas/headroom*) productizes the storage-layer problem this paper leaves partly open: it implements **Compress-Cache-Retrieve (CCR)**, a *lossless-under-recall* eviction discipline in which every original artifact is cached and the agent can retrieve the full version on demand whenever the compressed form proves insufficient. Headroom compresses *task-time tool output* within a single session – JSON arrays through a statistical compressor, code through tree-sitter AST awareness, logs and diffs through type-specific handlers – and reports 60–95% token savings with near-zero accuracy delta on GSM8K, TruthfulQA, SQuAD, and BFCL through a reproducible evaluation suite. Sleep Consolidation compresses a different target on a different cadence: *cross-session consolidated memory* (operational lessons, knowledge files, the never-delete recipe and skill archives) compacted nightly. The two are convergent on the same Compress-Cache-Retrieve principle at different timescales – within-session reversible CCR versus cross-session reversible

CCR – and Section 8.2 adopts Headroom's reversibility guarantee directly to close a hedge the storage layer currently carries.

Adversarial self-review and authoring discipline. Agent-Skills (Osmani, *addyosmani/agent-skills*) ships *doubt-driven development*: a structured CLAIM → EXTRACT → DOUBT → RECONCILE → STOP loop in which the artifact and its contract are extracted with the *original reasoning stripped*, then handed to a **fresh-context adversarial reviewer** that never saw the chain of reasoning that produced the artifact. This is directly relevant to the self-reinforcing error spiral of Section 8.1: the same model instance that wrote a mutation shares the priors that produced it and cannot be trusted to doubt it, whereas a reviewer fed only the artifact and the principles it must not contradict can challenge it without the self-confirming context. Osmani applies the loop to code claims within a single development session, bounded by user override; Section 8.1 adapts it to prompt, recipe, and strategy mutations across the nightly consolidation boundary, bounded by the existing manifest and human-review gate. Its two authoring conventions – a *When NOT to use* anti-trigger and a *Loading Constraints* declaration on every skill – map onto our depth selector and per-target gating, and are imported as design principles in Section 7.

Our contribution differs from all of the above in three ways: (1) it operates on *production* personal-assistant workloads, not benchmarks or simulations; (2) improvement is *cross-task* – lessons propagate through shared infrastructure; and (3) reflection is *scheduled and autonomous*, not triggered by explicit failure signals or human feedback.

3. Architecture – Sleep Consolidation

3.1 System Overview

Sleep Consolidation is not a single component. It is an emergent property of four interacting systems.

Nightly Cycle (reflection layer). A wind-down pass reviews the day and encodes lessons; a memory-consolidation pass compresses logs and routes knowledge; a cleaning pass prunes and enforces budgets. These run on a fixed nightly schedule (Section 3.4).

All three write to the **Shared Memory Layer:**

- an operational-lessons file (behavioral rules – the primary mutation target)
- a structured failure database (failure patterns with root-cause analysis)
- domain knowledge files (principles and procedures, sorted by topic)
- a core-principles file injected into every session's context window
- the cron prompts themselves (self-modifying task instructions)

Daytime Crons (execution layer, consumers and producers). A morning briefing, an online-engagement pass, a life-butler pass, and roughly a dozen others. Every cron job is both a *consumer* of the shared memory layer and a *producer* for it. It reads operational lessons, principles, and prior failures before acting; later it emits receipts, logs failures, and – during the nightly cycle – helps mutate the shared layer itself.

What makes the architecture effective is not mere persistence but purposeful arrangement. A chronological archive tells the system what happened; a use-case-sorted memory layer tells it what to do.

The three layers this meta-system tunes can be summarized self-containedly, and each is maintained by a specific nightly pass:

- **Storage layer.** Records every interaction and compacts older events into compressed pointer summaries. The memory-consolidation pass adjusts compaction aggressiveness

based on observed retrieval patterns – if important memories are being over-compressed, the thresholds loosen. The discipline this layer aims for is reversible Compress-Cache-Retrieve (Chopra, *chopratejas/headroom*): a compacted summary should carry a handle back to the cached original so that compression is reversible by construction rather than lossy, and Section 8.2 takes this from aim to contract.

- **Retrieval layer.** Maps keywords and embeddings to memory segments. A dedicated index-rebuild pass refreshes it nightly and can adjust scoring weights – for example, boosting a domain's memories after repeated retrieval failures there.
- **Identity layer.** Reads prompt files (persona definitions, behavioral rules, operational lessons) to keep personality and behavior consistent. The wind-down pass rewrites those same files, creating the core feedback loop: behavior produces incidents, reflection reads incidents, prompt mutation changes the files the identity layer consumes, and tomorrow's behavior shifts accordingly.

The mapping is concrete: the index-rebuild pass maintains the retrieval layer, the consolidation pass maintains the storage layer and routes new knowledge, and the wind-down pass maintains the identity layer by mutating its prompt files. Appendix A lists the full cron schedule and which tier of model runs each pass.

3.2 Algorithm 1: The Nightly Reflection Loop

```
ALGORITHM 1: Nightly Sleep Consolidation Loop
==========================================
INPUT:  daily_log (all cron receipts + interactive session logs from past 24h)
        operational_lessons (current shared behavioral rules)
        bug_reports (structured failure database)
        cron_prompts (current task-specific instructions)
OUTPUT: operational_lessons' (updated rules)
        bug_reports' (new/updated failure patterns)
        cron_prompts' (mutated instructions)
        mutation_log (what changed and why)

PHASE 1: WIND-DOWN (model: strongest available)
  incidents = extract_failures(daily_log)
  FOR EACH incident IN incidents:
    root_cause = classify(incident):
      EXTERNAL  -> log_only(incident); CONTINUE
      MODEL_CAP -> log_with_tier(incident); CONTINUE
      PROMPT_GAP -> GOTO MUTATION

    MUTATION:
      delta = minimum_prompt_change(incident, cron_prompts)
      IF contradicts(delta, operational_lessons):
        flag_for_human_review(delta)
      ELSE:
        apply(delta, target=cron_prompts OR operational_lessons)
        log_mutation(delta, incident, target)

      depth = fractal_depth_selector(incident)
      IF depth >= 2:
        meta_lesson = abstract_principle(incident)
        IF meta_lesson.scope == SYSTEM_WIDE:
```

```
          append(meta_lesson, operational_lessons)
        IF depth >= 3:
          flag_for_human_review(meta_lesson)  // Level 2+ gate

PHASE 2: CONSOLIDATION (model: strongest available)
  compress(daily_logs older than 3 days)
  route_knowledge(new lessons -> knowledge subdirectories)
  rebuild_search_indexes()
  build_concept_index(modified knowledge files)   // see Section 6.4

PHASE 3: PRUNING (model: mid-tier)
  enforce_size_budgets(operational_lessons, max=50KB)
  archive_stale(lessons not referenced in 14 days)
  prune_sessions(inactive > 48h)
```

3.3 The Three Mechanisms

Mechanism 1: Failure-Driven Prompt Mutation

When an agent hits a failure, the default response is to log it and move on. Sleep Consolidation adds a second step: **determine whether the failure came from a prompt deficiency, and if so, mutate the prompt.** The agent rewrites its own operating manual each night – like a pilot who, after every flight, updates a personal checklist based on what went well and what nearly went wrong.

Production example – The B010 cascade (Mar 3 to Mar 10):

This seven-day sequence shows the full loop, including second-order correction.

Day 0 (Mar 3) – Initial failure. A fork-sync cron job, prompted to "sync the fork with upstream," read that instruction as permission to edit production source code, run builds, and restart the gateway. The problem was not model incapacity. It was prompt ambiguity.

Prompt before: "Sync the fork with upstream and report the result." *Prompt after (mutation #8):* "Run the safe merge script. NEVER modify source code directly. NEVER run builds. NEVER restart the gateway. If the script fails, report and STOP."

The mutation was filed as bug B010, root cause: "Ambiguous action verb ('sync') interpreted as permission for unrestricted operations."

Day 7 (Mar 10) – Second-order failure. The same cron later hit a merge conflict. It reported the conflict and stopped – exactly as mutation #8 required. But the restriction was too broad. The agent was in fact *capable* of resolving the conflict by reading the fork's patch documentation, understanding both sides' intent, and verifying with a build. The blanket prohibition prevented useful work. The merge was blocked for 24 hours.

Prompt after (mutation #14): "Run the safe merge script. If conflicts remain, read the patch registry to understand intent, attempt resolution, verify with build. If genuinely uncertain, escalate."

Meta-lesson encoded (Level 2): "When encoding a safety lesson, separate the failure mode from the restriction. The restriction should be proportional to the risk, not a blanket prohibition."

That meta-lesson now applies to *all* future lesson encoding across the system. One incident produced a permanent improvement in how the system learns. As Section 6.2 shows, this same cascade is now the canonical test case for an automated strategy-switch trigger.

Mechanism 2: Fractal Depth Calibration

Not all tasks deserve the same metacognitive effort. Sleep Consolidation includes a **depth selector** – a prompt-encoded heuristic that decides how many layers of "why" to traverse:

Signal	Depth	Example
Routine / mechanical	0	Read a file, send a message
Something broke	1–2	Why did it break? What pattern?
Encoding a new rule	2–3	Is the abstraction right? Am I over-correcting?
Explicit request for depth	3+	Go until insight stops being actionable

The self-similar property: the depth selector is itself subject to refinement. When the agent zooms too deep on trivia and wastes tokens, or too shallow on something important and misses the lesson, that miscalibration becomes training data for adjusting the selector.

Convergence hypothesis (observed, not proven): Most tasks stabilize at depth 0–1 within days of operation, while novel failure classes begin at depth 2–3 and move to 0–1 as their patterns become encoded. We see roughly weekly Level 2 refinements and roughly monthly Level 3 refinements, which suggests deeper reflection becomes less necessary as the system matures. We do not claim formal convergence – only that the 30-day pattern is consistent with it.

Production example – Fractal emergence (Mar 10): The merge-conflict episode began at depth 0: just do it, abort on failure. After the B010 over-correction, a human prompt ("think fractal") triggered depth 3+ analysis, which revealed that the real problem was not merge strategy but the *method used to encode lessons.* The resulting principle – "separate the failure mode from the restriction" – now applies to every future lesson, not just merge conflicts.

Mechanism 3: Cross-Cron Knowledge Transfer

The strongest emergent property is that lessons learned in one autonomous task propagate to all others through shared memory files. A lesson learned in one domain – "always double-check dates" – spontaneously improves performance in another, the way a surgeon's steady hands might also make for a better painter, though no one taught the transfer. We documented 5 cross-domain transfer events over 30 days:

#	Origin Task	Lesson Encoded	Transferred To	Observable Behavior Change
1	Calendar management	"When 'the X' is ambiguous, ask before acting"	Fork sync	Cron reported two interpretations of a conflict instead of guessing
2	WhatsApp messaging	"Use ID-based targeting, never name-based"	All crons with messaging	Morning briefing switched from name to ID targeting
3	Privacy incident	"Never include phone numbers in visible text"	All sessions	Online engagement cron redacted numbers in PR comments
4	Fork sync (B010)	"Report-only crons must not modify systems"	Security audit cron	Audit reported findings without attempting fixes

#	Origin Task	Lesson Encoded	Transferred To	Observable Behavior Change
5	Memory consolidation	"Enforce file size budgets"	All report-generating crons	Reports began self-truncating at budget limits

The transfer mechanism is architectural, not magical: lessons written to the operational-lessons file are read by all sessions at boot. But the **abstraction level** at which lessons are encoded determines transfer breadth. "Use ID-based targeting" (Level 0, specific) transfers only to messaging tasks. "When something is ambiguous, ask before acting" (Level 1, general) transfers much more widely.

Cross-cron transfer depends less on storing more and more on storing in the right shape: partitioned by decision type, action domain, and operating rule rather than accumulated chronologically.

3.4 The Nightly Cycle

The cycle runs as an ordered sequence of passes, each producing inputs for the next:

Order	Pass	Role	Biological Analogy
1	Wind-Down	Review, identify failures, encode lessons, mutate prompts	Sleep replay / error correction
2	Memory Consolidation	Compress daily logs, route knowledge, rebuild indexes	Hippocampal-to-neocortical transfer
3	Fork Sync	Integrate external changes, self-heal build failures	Environmental adaptation during rest
4	Cleaning	Trim bloated files, prune stale sessions, enforce budgets	Synaptic pruning

Critical ordering: Wind-Down runs first because it produces the raw material – identified failures and encoded lessons – that Consolidation then routes to permanent storage. Cleaning

runs last because it prunes temporary artifacts. We learned the cost of getting this wrong when a misconfigured schedule ran the cleaning pass before Wind-Down, pruning the very daily logs Wind-Down needed as input. (Wall-clock times for each pass have shifted across the deployment's life; the *ordering* is the invariant, not the hour.)

4. Production Evidence – 30 Days of Observed Improvement

4.1 Methodology and Limitations

System: an OpenClaw-based personal assistant running on a Linux workstation. Primary reflection model: the strongest available frontier model; daytime crons use a mix of strong and cheaper models. Roughly 12–13 autonomous cron jobs over the period. One human operator.

Period: February 8 – March 10, 2026 (30 days).

Data sources: cron receipts (structured JSON, auto-generated per run), the operational-lessons file (git-versioned, all changes tracked), bug reports (structured markdown, manually filed), and daily Wind-Down logs.

Labeling protocol: error instances were identified from cron receipts (non-zero exit codes, escalation flags) and operator-reported incidents. Error *classes* were defined post hoc by the authors based on root-cause similarity. No inter-rater reliability was computed – this is a single-system observational study, not a controlled experiment.

Confounds we acknowledge: over the 30-day period, the following changed at once: prompt content (the variable under study), cron ordering (adjusted twice), model assignments (three crons moved to a cheaper model), operator familiarity, and the upstream codebase (109 commits merged). We cannot isolate the contribution of prompt mutation from these confounds. The evidence is *consistent with* the Sleep Consolidation hypothesis but does not *prove* it.

What a controlled study would look like: run two identical agent deployments, one with nightly reflection enabled and one with static prompts. Measure error-recurrence rate, human-intervention frequency, task-completion quality (rated by blind evaluators), and time-to-resolution for novel incidents. Duration: at least 4 weeks. We have not conducted this study.

4.2 Prompt Mutations Observed

Over 30 days we documented 14 prompt mutations – changes to cron prompts or operational rules triggered by real failures. "Documented" means the mutation has a git commit, a linked bug report or incident description, and a before/after prompt diff. We observed no mutations that were *reverted* as harmful, though mutation #8 was later *refined* by mutation #14 after an over-correction.

#	Date	Trigger	Mutation	Scope
1	Feb 10	Morning briefing missed calendar events	Added dual-calendar query requirement	Single cron
2	Feb 14	Cron sent message to wrong chat	Added explicit ID-based targeting, banned name lookup	All messaging crons
3	Feb 16	Memory files growing unbounded	Created size budgets, cleaning cron	System-wide
4	Feb 19	Agent skipped its boot-context file	Added mandatory boot sequence	All sessions

#	Date	Trigger	Mutation	Scope
5	Feb 22	Renamed function broke heartbeat calls	Created a patch registry	Fork sync
6	Feb 24	Cron report was 40KB (unreadable)	Added budget fuses, formatting rules	All report crons
7	Mar 01	Agent included PII in a reply	Added privacy guardrail	All sessions
8	Mar 03	Fork sync cron edited production code (B010)	Added hard constraints to cron prompt	Fork sync
9	Mar 03	Same session: cron killed the gateway	Added “never force-kill processes” rule	All crons
10	Mar 04	Build failed after merge – wrong externals	Auto-heal retry + wiring guardian	Fork sync
11	Mar 05	Model misidentified person from a headline	Added “always click through” rule	Self-evolution
12	Mar 07	Wind-Down wrote to the wrong day’s log	Added a temporal-awareness step	Wind-Down
13	Mar 08	Agent didn’t recognize a family member	Added mandatory daily-log reading to boot	All sessions
14	Mar 10	Merge abort on a single conflict too aggressive	Intelligent resolution with safe defaults	Fork sync

A mutation taxonomy. Sorting the 14 mutations by intent lets others prioritize which kinds to automate first:

Category	Mutations	Count	What it changes
Safety / guardrail	#2, #7, #8, #9	4	Prevents harmful or out-of-mandate actions
Completeness	#1, #4, #12, #13	4	Stops the agent acting on partial context
Resource / efficiency	#3, #6	2	Caps cost, size, and unreadable output
Resilience / recovery	#5, #10, #14	3	Self-heals after an external or build failure
Verification	#11	1	Forces a check before asserting a fact

Two patterns stand out. First, safety and completeness dominate early failures (mutations 1–9), while resilience and proportional-constraint refinements appear later (10–14) – the system first stops doing harmful things, then learns to keep doing useful ones safely. Second, the categories map cleanly onto the autonomy gates in Section 6: safety mutations are the ones that most need human review, while efficiency mutations are the safest to auto-apply.

4.3 Error Class Tracking

We tracked six error classes, defined by root-cause similarity:

Error Class	Definition	Wks 1–2	Wks 3–4	Status
Wrong chat / recipient	Message delivered to unintended target	3	0	No recurrence after mutation #2
File size explosion	Output exceeding readability / budget limits	2	0	No recurrence after mutation #3
Missing context at boot	Agent unaware of available information	4	0	No recurrence after mutations #4, #13
Unsafe cron actions	Cron acting outside its mandate	2	0	No recurrence after mutations #8, #9
Ambiguous request mishandled	Acting on ambiguous input without clarifying	3	1	Declining; 1 residual instance
Over-broad safety rules	Safety constraints blocking legitimate work	0	2	New class, found via fractal analysis

Total tracked incidents: 14 (weeks 1–2) to 3 (weeks 3–4), a 79% reduction. Mistakes here don't just get patched; whole classes get eliminated – like a spelling error that autocorrect learns once and never surfaces again. The emergence of "over-broad safety rules" as a *new* class in weeks 3–4 is especially notable. The error surface shifted from execution failures to policy failures. That is consistent with a maturing system: once first-order mistakes drop, second-order distortions become easier to see.

Caveat: we cannot cleanly separate "errors eliminated by prompt mutation" from "errors eliminated by operator learning" (the operator stopped triggering certain edge cases as he learned the system). The confound is real and uncontrolled.

4.4 Token Economics

Component	Tokens / night	Cost / night (est.)
Wind-Down	~15K	$0.45

Component	Tokens / night	Cost / night (est.)
Consolidation	~20K	$0.60
Cleaning	~8K	$0.12
Total nightly overhead	~43K	$1.17

Over 30 days, total reflection cost was about $35. We estimate 8 human-intervention incidents were avoided (based on error-class extinction – incidents that would otherwise have required the operator to debug and fix manually). Valuing operator time at $25/hr with a 30-minute average resolution yields about $200 saved.

Sensitivity analysis: at $15/hr (low) ROI is 3.4x; at $50/hr (high) it is 11.4x. If only 4 incidents were truly avoided (conservative), ROI at $25/hr is still 2.9x. Under every reasonable assumption the reflection loop is cost-positive, though the estimates are approximate.

5. The Fractal Metacognition Framework

5.1 Why Fractal?

"Fractal" here means **self-similar cognitive patterns applied at different scales of abstraction.** The same reflective sequence – *what happened, why, what principle, what meta-principle* – recurs at every level of task analysis.

The key insight: **the same reasoning that improves a specific task can also improve the category of tasks, the method used to improve categories, and eventually the method used to improve methods.** Each level operates on a smaller, more abstract domain than the one below.

5.2 Convergence Intuition (Not a Proof)

Consider error classes at multiple scales:

- **Level 0** (execution): fix the specific bug. Eliminates one instance of one class.
- **Level 1** (pattern): encode a rule preventing the class. Eliminates all instances of one class.
- **Level 2** (methodology): improve how rules are encoded. Reduces the rate at which new classes arise from over- or under-correction.
- **Level 3** (meta-methodology): improve how improvement works. Reduces the overhead of reflection itself.

Each level operates on a smaller domain (instances > classes > methodology > meta-methodology). We *hypothesize* this produces convergent behavior: the frequency and magnitude of changes should fall at each higher level. Our 30-day observations fit that pattern (daily Level 0–1, weekly Level 2, monthly Level 3), but 30 days is not enough to establish convergence. The hypothesis is falsifiable: if Level 2+ mutation frequency does not fall over time, the convergence claim fails.

5.3 The Depth Selector as a Learned Policy

How deeply to reflect is itself a learnable policy. Initially the agent has little calibration. Over time the depth selector improves through its own mistakes:

- *Reflected too shallowly* on a merge strategy –> missed the over-correction pattern –> added depth for "encoding new rules"

- *Reflected too deeply* on a routine file operation –> wasted compute –> reduced depth for "routine / mechanical"

We observe this self-calibration emerging over the 30-day period, but we lack quantitative metrics for "reflection-depth appropriateness." An instrumentation layer tracking tokens-spent-on-reflection versus value-of-insight-produced would enable a more rigorous evaluation.

6. From Declarative to Procedural: Evolving Recipes, Strategies, and Skills

Sections 3–5 describe consolidation of *declarative* memory – rules and facts. The same nightly loop has since been extended to *procedural* memory: the agent's own recipes, the strategies it switches between, and the reusable skills it distills. This is the cerebellum half of the analogy made literal – offline practice refining the *procedures* themselves, not just the rules about them. All of these extensions share one shape: **observe the day's episodes, distil a durable ranked artifact, feed it back into tomorrow's behavior.** Each is an opt-in step in the consolidation pipeline, off by default, so the loop's baseline behavior is unchanged when the extension is absent. The three primary extensions operate at the recipe, strategy, and skill level; a finer per-step sharpening signal (Section 6.5) practices the single step of a recipe that keeps failing.

A load-bearing design boundary keeps this honest. Recipe *execution* lives in a separate subsystem; consolidation does not own it and must not duplicate it. The contribution here is strictly the *offline learning loop*: read episode outcomes, compute fitness, propose mutations, and write a ranked archive that the execution-time selector consumes. Recipes are owned by the execution subsystem; their *measured fitness* is owned by consolidation. Any write back into the execution subsystem's recipe directory is a *proposal* gated by human review (Section 7.1), never a silent overwrite.

6.1 Recipe Evolution: Empirical Fitness and a Never-Delete Archive

A recipe is a procedure the agent runs to accomplish a recurring task. Recipes have historically been hand-written and static; nothing measured whether one was better than another, and nothing recorded how a recipe reached its current form. The recipe-evolution step adds a selection loop over the recipe population. Each night it attributes every completed episode to the recipe it used, scores that recipe's *empirical fitness* (success rate, latency, token cost), proposes mutations (add, remove, or reorder a step; tighten or loosen success criteria), keeps a never-delete archive of versioned variants, and ranks variants so the selector can prefer winners. It is the cerebellum turning a fumbled piano piece into a fluent one over successive nights.

Fitness is computed from episode outcomes – a `completed` episode counts as a success, an `abandoned` one as a failure – with the success rate Laplace-smoothed so that one success out of one run is not reported as a perfect 1.0. Latency is the episode's end-minus-start time; token cost sums the events in the episode. The hard part is *attribution*: tying an episode to the recipe that ran it. Without a reliable recipe tag on each episode, fitness is computed against the wrong recipe and the loop learns garbage. The implementation only attributes when an explicit `recipe:<slug>` tag is present and returns no attribution otherwise, refusing to guess.

Attribution is worth describing as a closed cycle because it is the seam where an offline loop most easily goes inert. The tag needs a *producer* as well as a consumer: at dispatch time a single `recipe:<owner/slug>` marker is appended into the run's episode event store – a `turnId:0` system event shaped so it can neither flip the episode's inferred outcome (which keys off the *last* event) nor fragment the run into multiple episodes, and best-effort by design so a failed

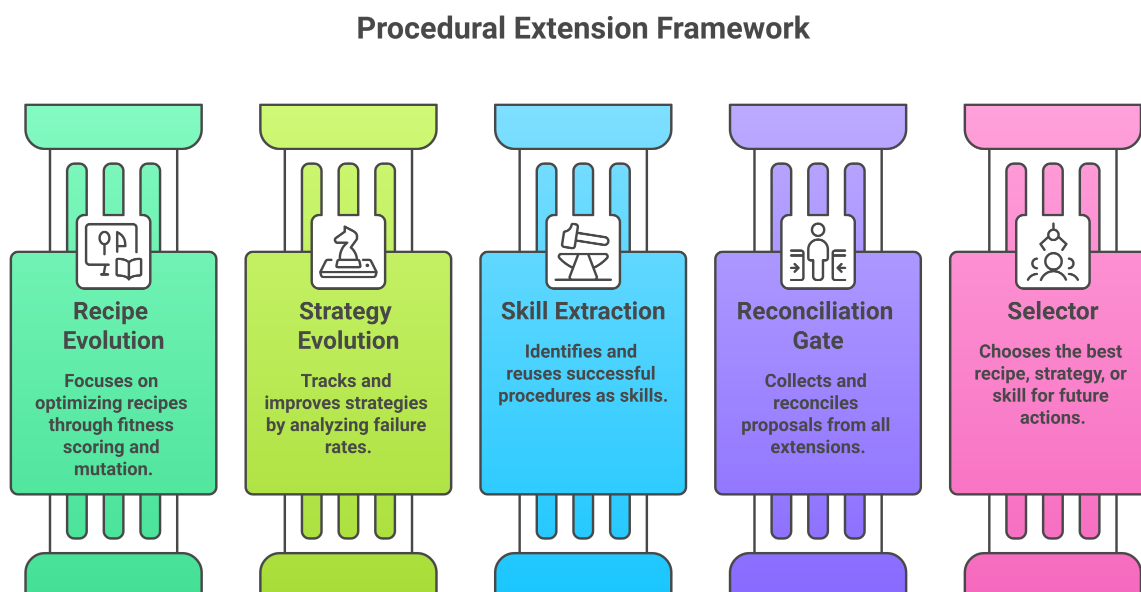

Figure 2. The three procedural extensions share one pipeline: each night the agent observes the day's episodes, distils a durable ranked artifact (recipe fitness, strategy-failure counts, extracted skills), and a single gate reconciles all proposals into one human-reviewed manifest that feeds tomorrow's selector.

append never breaks the run it annotates. The nightly pass reads those tagged episodes, builds a fitness record keyed by the full `owner/slug`, and the *same* key is looked up at execution time to modulate runtime effort: a low-fitness recipe is treated as uncertain and given more re-dispatch and recovery headroom, a high-fitness one less. So measured fitness is not only an offline selection signal for ranking variants; it also shapes how much budget tomorrow's run receives. The load-bearing correctness property is the one-key round-trip – the `owner/slug` produced must equal the `owner/slug` the consumer scores under – because keying a lookup by the wrong identifier (slug-only against a full `owner/slug`) is exactly the silent mis-credit attribution is meant to prevent.

Proposals are gated, not applied. They are written to a daily manifest for review. A narrow auto-promotion path exists for the safest case – a *corrective* proposal (driven by a success rate at or below half the floor, i.e. 0.25 against a 0.5 floor, not merely under it) that is well-evidenced (at least 8 runs, above the 3-run minimum needed merely to propose) and reversible (the never-delete archive keeps every prior variant). Efficiency proposals, such as those driven by latency regression, are never auto-promoted. This is autonomy-first but bounded, and it maps directly onto the mutation taxonomy of Section 4.2: efficiency-class changes are the only ones the system trusts itself to apply unsupervised, and even those are logged and reversible.

```
interface RecipeFitness {
  recipeId; version; runs; successes;
  successRate;              // Laplace-smoothed for low n
  avgLatencyMs; avgTokenCost; lastUpdated;
}
interface MutationProposal {
  recipeId; baseVersion;
  op;                       // add_step | remove_step | reorder | tighten | loosen
  payload; rationale; expectedDelta;
  needsHumanReview;         // false only for corrective + well-evidenced + reversible
```

```
}
// nightly step (opt-in)
for each completed episode:
  rid = attributeRecipe(episode)        // null unless a recipe:<slug> tag is present
  if rid is null: continue
  fit = updateRecipeFitness(archive.latest(rid), episode)
  archive.putVariant(rid, fit.version, currentBody(rid), fit)
  proposals += proposeMutations(fit, archive.history(rid))
writeManifest(proposals)                // gated; auto-promote only the safe subset
```

6.2 Failure-Count Strategy Switch

A single failure triggers a prompt mutation, but the early system had no memory of *how many times in a row* a given strategy had failed, and no rule that said "this approach keeps failing – switch to a different one." The strategy-switch step adds a per-strategy state machine that counts consecutive failures and fires a switch after a threshold is crossed within a recency window. This is the restaurant that, after the risotto comes back three nights running, stops tweaking the recipe and takes it off the menu.

The canonical case is the B010 cascade from Section 3.3. Three consecutive failures of an always-merge fork-sync strategy should trip a switch to an ask-before-merge strategy – not a fourth identical patch. The implementation makes this concrete: the default threshold is **3 consecutive failures**, the recency window is **24 hours** (failures spread beyond it do not count as a pattern), and a switch is flagged for human review unless confidence clears **0.8** and a registered fallback strategy exists. Confidence rises with the overshoot past the threshold, so a strategy that has failed five times running is trusted more than one that has just reached three.

Two failure modes shaped the design. First, episode-boundary false positives: a long failing task can be split by the time-gap detector into several `abandoned` episodes, over-counting consecutive failures and tripping a premature switch – so failures within the same task are de-duplicated. Second, idempotency: if the consolidation cursor goes stale and events are reprocessed, failures would double-count, so records are keyed by event id. A successful episode resets the counter to zero and, if a switch was just made, stamps the time-to-recovery – which makes a switch that *didn't* help visible rather than silently sticky.

```
interface SwitchDecision {
  shouldSwitch; fromStrategy; toStrategy;   // toStrategy null -> no fallback -> human
  confidence; needsHumanReview; rationale;
}
// decideSwitch:
//   IF consecutiveErrors >= threshold (default 3)
//      AND (now - lastFailure) < windowMs (default 24h)
//   THEN switch to fallback if registered, else flag for human;
//        needsHumanReview = confidence < 0.8 OR no fallback
//   ELSE no switch
```

The open contract decision – whether the fallback map is hand-authored per strategy or learned from which alternatives historically scored higher fitness (which would couple this step to the recipe archive of 6.1) – is deliberately left to a human. So is whether `abandoned` is a sufficient failure signal, since an episode can be abandoned because the *user* walked away, not because the strategy failed.

6.3 A Skill Library, Voyager-Style

Voyager's idea: as an agent solves tasks, distil the successful behavior into a named, reusable skill, store it in a growing library, rank skills by success, and retrieve and compose them on future tasks so competence compounds instead of being re-derived. The skill-extraction step applies this to a general assistant workload. From each *completed*, tool-using episode with a clear decision, it synthesizes a skill – name, description, prerequisites, steps, test cases, success metrics – stores it versioned in a never-delete library, ranks by empirical success rate, and exposes it to retrieval so a recurring situation can carry "this previously used skill X with Y% success." It is the cross-cron transfer of Section 3.3 turned from an emergent side effect into a first-class, addressable artifact.

Three guards keep the library from rotting. A strict skill-worthiness gate – completed plus a tool call plus a multi-step decision – refuses to learn skills from trivial chat or abandoned attempts, so the library does not fill with non-general entries. A Jaccard-similarity dedup at **0.8** over the skill body merges near-identical skills rather than spawning duplicates; when a same-named skill recurs it becomes a new version, and the source-episode ids are merged. And success metrics decay on recent failure so a skill encoding an obsolete procedure stops ranking on past glory and becomes reachable for deprecation. Deprecation marks but never deletes – the body is still readable, preserving the never-delete invariant the rest of the system relies on.

The "skill-as-code versus skill-as-procedure" question this design raises has a concrete answer in this deployment: a skill is a typed record – a name, description, prerequisites, ordered steps, and optional input/output schemas, plus a flat `lineage` field recording whether it came from composition, episode extraction, or promotion. It is invoked *inline* from a recipe through an `invoke skill:<id>` step directive that sits alongside the existing sub-recipe and loop directives; the directive injects the skill's procedure into the task and fails closed if the named skill is absent or deprecated. A small seeded standard library of four typed primitives – summarize-text, extract-json-field, web-search-and-cite, and classify-text – ships as the bootstrap population so the library is never empty on a cold start. The deposit path is fail-closed in the same spirit: a malformed skill (one needing a non-empty name and at least one step) is rejected, same-name deposits version-bump rather than clobber, curated seeds are guarded against silent overwrite, and a `promote` flag applies a *live-margin* fitness bar – a candidate's measured success rate must clear the library's current success-rate mean plus one standard deviation (never a frozen N; an empty library is permissive). This is the "rank by empirical fitness, never delete" discipline made executable, governed by the same salience-not-frozen-bounds rule the rest of the loop follows.

Retrieval is genuinely semantic rather than string overlap: skill search resolves the same in-process embedding function the consolidation pass uses (one embed provider, not a parallel one), does a batched-embed plus cosine ranking, and falls back to keyword matching only when no provider is present. When a new task finds no matching recipe, the system tries *compose-from-library* first – a deterministic step that turns a skill search into a runnable recipe, one `invoke skill:` step per hit in rank order, authored as a non-curated "on-the-fly" recipe with its lineage stamped into the frontmatter so the authorship guard permits it without clobbering any hand-written recipe. Competence compounds instead of being re-derived, which is the Voyager promise this section invoked.

One honest gap remains at the consolidation seam. Skill *deposit, retrieval, and invocation* are live, but the path by which the *nightly loop itself* mints new skills from the day's episodes is not yet wired: episode detection emits an empty key-decision list, so the skill-worthiness gate – correct in itself – is starved of input and the live library grows only from seeding and explicit deposits, not organically overnight. Closing that requires deriving key decisions from each episode's own tool-call trace (two or more distinct tool actions marking a multi-step procedure worth extracting; a lone one-shot call declined). Until that producer is added, the flywheel turns on demand but not in its sleep – the same producer-seam lesson that the recipe-fitness loop already learned.

6.4 One Gate for Three Mutation Sources – and a Consolidation Output the Thesis Demanded

All three extensions add new mutation or selection targets, and on a busy night they can fire for the same task at once: "mutate recipe X," "switch strategy," "use skill Y." Three ad-hoc gates would be three audit surfaces and three chances to thrash. The design calls for **one** arbitration point that collects the night's proposals, de-duplicates conflicting targets for the same task, applies a fixed precedence (a strategy switch outranks a skill suggestion, which outranks a recipe mutation, because a switch can invalidate the recipe a mutation was tuning), and writes a single reconciled manifest for review. This keeps the human-in-the-loop principle (Section 7.1) intact behind one surface. The per-night manifest that collects every proposal is in place; the precedence-based reconciler that de-duplicates conflicting targets for a task is the design's next step, not yet shipped.

These procedural extensions also exposed a gap in the *declarative* loop that the paper's own thesis predicted. The thesis says consolidation must organize memory by purpose, not chronology. The early implementation did this for lessons, preferences, and contacts – but not for *concept-to-document routing*. On April 1, a memory search returned the wrong documents because no concept index existed to route a topic to the files that discuss it. The system meant to prevent retrieval failure through organization had failed to organize the retrieval-critical artifact. The fix is a consolidation output (shown in Algorithm 1, Phase 2): for each knowledge file modified that day, extract retrieval keywords – including the synonyms a human would actually search for – and write a concept index mapping keywords to file paths. New or substantially changed knowledge files are also seeded with retrieval keywords so the keyword channel has material to match even when the document's natural language does not contain the search terms. We record the April 1 incident as the consolidation system's own failure case, which is exactly the kind of negative result Section 7 argues should be reported rather than hidden.

6.5 Per-Step Struggle Sharpening

Recipe fitness (Section 6.1) is *coarse*: it scores a whole recipe pass/fail and proposes whole-recipe mutations. But a recipe usually does not fail uniformly – it fails at one step. A finer signal asks not "is this recipe weak?" but "which bar of the piece does it keep fumbling?" – and practices that bar, not the whole piece. This is closer to what motor consolidation actually does, and it is the cerebellum analogy made one level sharper.

The mechanism reads a source the coarse fitness store does not: the *live plan archive*. Every recipe run is archived as a plan with a per-step `status` and, on failure, a classified-error envelope. Across a recipe's archived runs, the reader aggregates each step's failure rate and the *modal* classified-error kind for that step, then flags steps that fail noticeably worse than the recipe's own baseline. For each flagged step it proposes a new mutation operator, `rewrite_step_text` – sharpen the prose of that single struggling step, one step body changed, a patch-version bump – applied through the same snapshot-reversible, authorship-guarded, validate-or-skip recipe-write path the other extensions use. It re-measures by diffing the step's failure rate across a later archive pass: a true before/after on the *step*, not a self-confirming success count.

Two properties keep it disciplined. First, it routes around an inert store on purpose: it reads execution traces the system already keeps for free, so it produces a signal even when the recipe-fitness store is empty. With both signals present the loop now has two complementary fitness views – coarse recipe-level and fine per-step – not one redundant pair. Second, its thresholds are derived from the live sample, not frozen: the minimum-failure floor before a step is worth sharpening grows with the square root of the step's run count, and the struggle threshold tracks 1.5x the recipe's own mean step-failure rate, clamped. This is the same no-frozen-constants governance the rest of the consolidation loop follows, applied to the procedural side.

Its default posture matches the other extensions: it *proposes only*. Application is gated behind

an explicit auto-apply flag; only a step rewrite that is reversible, well-evidenced, and dominated by a recoverable error kind is flagged auto-promotable, inheriting the human-in-the-loop default of Principle 7. A nightly lane dispatches the optimize pass for struggling recipes, and the absence of any archive leaves the consolidation output byte-identical – the same opt-in, safe-when-absent discipline every Section 6 extension claims.

7. Design Principles for Sleep Consolidation Systems

From production experience, we propose nine design principles. The last two are imported from external authoring discipline (Osmani, *addyosmani/agent-skills*) rather than derived from our own incidents, and are flagged as such.

Principle 1: Separate Fast and Slow Learning

Operational sessions handle tasks (fast). Nightly crons handle reflection (slow). Never mix reflection into task execution – it degrades task performance and produces worse reflection. This mirrors Kahneman's System 1 / System 2 distinction (Kahneman, 2011) and the biological separation of online performance from offline consolidation.

Principle 2: Encode at the Right Abstraction Level

"Always check both calendars" is Level 0. "When a system has multiple data sources, query all of them" is Level 1. "Verify completeness assumptions before acting" is Level 2. Encode at the highest level that stays *actionable* – too abstract and it becomes empty, too specific and it fails to transfer.

Principle 3: Shared Memory > Private Memory

Lessons stored in a single cron's prompt help only that cron. Lessons stored in shared operational files help every session. Default to shared storage; use private storage only for domain knowledge that would confuse other tasks.

Principle 4: Proportional Constraints

When encoding a safety lesson from a failure, the restriction should match the risk. "Never edit source code" is disproportionate to "a cron edited the wrong file once." "Understand intent before editing, verify after, revert if broken" is proportional and preserves capability. This is the central lesson of the B010 –> mutation #14 cascade, and it is now enforced mechanically by the strategy-switch step (6.2): a repeated failure switches the strategy rather than piling on another blanket prohibition.

Principle 5: Temporal Ordering of Nightly Cycles

Reflection –> Consolidation –> Integration –> Pruning. Each stage produces inputs for the next.

Principle 6: The Budget Fuse

Every reflection loop must have a token budget and a wall-clock timeout. Without one, a sufficiently thorough reflection agent will chase diminishing returns indefinitely. The procedural extensions of Section 6 inherit this discipline: proposals require a minimum number of runs before

firing, and the skill library carries the same size-budget and dedup pressure the operational-lessons file already does.

Principle 7: Human-in-the-Loop for Level 2+ Changes

Level 0–1 mutations (specific fixes and pattern rules) can be autonomous. Level 2+ mutations (methodology changes, core-principle edits) must be flagged for review. **Concrete gate:** any mutation that modifies the core-principles file (injected into all sessions) or changes the depth selector itself requires human approval before taking effect. Mutations to individual cron prompts or the operational-lessons file can be auto-applied but are logged with full diffs for audit. The procedural extensions follow the same rule: recipe mutations, strategy switches, and skills are gated by default, with a narrow auto-promotion path only for corrective, well-evidenced, reversible changes (Section 6.1).

Rollback mechanism: all mutations are git-committed with descriptive messages and can be reverted. Each night logs a "mutation manifest" listing all changes, enabling batch review; the procedural extensions write to the same manifest so there is a single audit surface.

Principle 8: Encode an Anti-Trigger, Not Just a Trigger

A lesson that only says *when* to fire is half a rule. Borrowed from the "When NOT to use" convention that every Agent-Skill carries (Osmani, *addyosmani/agent-skills*), this principle says each encoded lesson, recipe, and skill should also declare the conditions under which it must *not* apply. This is the depth selector made explicit: the selector already decides when *not* to reflect deeply, and the same discipline belongs on every artifact the consolidation loop writes. An over-broad safety rule – the very error class that emerged new in weeks 3–4 (Section 4.3) – is precisely a rule with a missing anti-trigger; encoding the negative case up front is the structural prevention for it, not just the post-hoc fractal correction.

Principle 9: Declare Where a Mutation May Load

Borrowed from the "Loading Constraints" convention of Agent-Skills (Osmani, *addyosmani/agent-skills*), every mutation should declare its load scope explicitly – which files it may touch and which sessions it may reach – rather than leaving scope implicit in the act of writing. This is the structural form of Principle 7's per-target gating: a mutation that declares "core-principles file, all sessions" is self-identifying as Level 2+ and routes itself to human review, while one that declares "a single cron prompt" is self-identifying as auto-applicable. Making the load scope a declared field rather than an inferred property is what lets the Section 6.4 gate arbitrate by precedence without re-deriving each proposal's blast radius.

8. Limitations and Failure Modes

8.1 The Self-Reinforcing Error Spiral

If a reflection loop encodes the wrong lesson, that lesson shapes future behavior, generating data that appears to confirm it – the AI analogue of confirmation bias. The risk grows with the procedural extensions of Section 6: a bad recipe mutation or an over-eager strategy switch could generate episodes that look like vindication.

Mitigations (implemented):

- Human review for Level 2+ changes (Principle 7), extended to recipe mutations, strategy switches, and skills by default.

- Git-versioned mutation log and a never-delete archive enabling revert at any point.
- A nightly mutation manifest – one surface – for batch review.
- Minimum-run thresholds before any procedural proposal fires, so low-n noise cannot trigger a change.

Mitigations (proposed, not yet implemented):

- Automated regression checklist: after each mutation, re-run known-good scenarios to detect degradation.
- "Canary mode": apply new mutations tentatively for 48 hours, then auto-revert if error rates rise.
- Signed mutation provenance recording the triggering incident, the model that generated the mutation, and a confidence level.
- A **fresh-context doubt pass**, adapting doubt-driven development (Osmani, *addyosmani/agent-skills*): before the single-gate reconciler of Section 6.4 commits the night's manifest, a reviewer in fresh context – fed only the *artifacts* (the mutation diffs, recipe-fitness records, strategy-switch decisions) and the *contract* (the operational-lessons principles they must not contradict), with the original reasoning that produced them deliberately stripped – adversarially challenges each proposed change. The mechanism design is precise about *why* it helps: the model instance that wrote a mutation shares the priors that produced it, so it cannot be relied on to doubt it; a reviewer that never saw that reasoning can. The pass is bounded the same way Osmani bounds it – trivial findings, a small cycle cap, and human override – and folds into the existing manifest plus human-review gate rather than adding a new audit surface. This directly attacks the spiral: a lesson that merely re-confirms its own priors is the failure mode, and a reasoning-blind reviewer is the cheapest available antidote.

8.2 Context Window Pressure

Every encoded lesson consumes context-window space. As the operational-lessons file grows, it competes with task-relevant context. The cleaning pass enforces size budgets (currently 50KB) and archives stale content. In the earlier design this created a second-order problem – archived lessons could become unavailable when they were relevant again, with only best-effort semantic retrieval to surface them, a hedge rather than a guarantee.

The principled fix is to make compaction *reversible by construction* rather than lossy, adopting the Compress-Cache-Retrieve (CCR) discipline of Headroom (Chopra, *chopratejas/headroom*). Under CCR, every compressed pointer-summary carries a handle to the cached original, so a lesson that proves insufficient in its compacted form can always be retrieved in full – *lossless under recall* – rather than relying on a re-embedding lottery to find it again. Headroom validates this within a single session, compressing task-time tool output and reporting 60–95% token savings with near-zero accuracy delta; Sleep Consolidation applies the same contract across sessions to consolidated memory, and the difference that matters is precisely reversibility: our archived lesson must never become unreachable. A second property worth borrowing is Headroom's per-content-type compressor family rather than one uniform compaction, which matters here because our knowledge files, JSON cron receipts, and recipe bodies have very different compressibility and lose very different things under naive truncation. With CCR in place, this section's hedge ("archived content may be unavailable") collapses to an engineering invariant rather than a residual risk. The skill library of Section 6.3 raises the same bloat risk and inherits the same budget, dedup, and never-delete discipline – which is itself CCR-shaped: originals kept, ranked variants surfaced.

8.3 Model Dependency

Reflection quality depends on the model's reasoning capability. We use the strongest available model for nightly reflection and cheaper models for daytime execution, which creates a cost asymmetry: the "learning" is only as good as the reflection model. A weaker reflection model would produce lower-quality mutations, potentially encoding bad lessons and amplifying the spiral of Section 8.1. The skill-extraction step adds a related risk – the same episode extracted twice with different wording can slip past string-level dedup – which is why semantic dedup, not just Jaccard, is the intended longer-term guard.

8.4 Single-Operator Bias

Our data comes from one operator with specific workflows, preferences, and failure patterns. Whether Sleep Consolidation generalizes to multi-user systems, other cultural contexts, or domains beyond personal-assistant tasks is an open question. We hypothesize that the *mechanisms* generalize (failure-driven mutation, cross-task transfer, depth calibration, recipe and skill evolution) even if the specific *lessons* do not.

8.5 No True Generalization

The fundamental limit is that prompt-mediated learning produces *declarative* improvements (better rules, better context), not *procedural* improvements in the model's own reasoning. The recipe and skill extensions of Section 6 push further into procedural territory than rules alone, but they still encode procedures *as artifacts the unchanged model reads*, not as better reasoning inside the model. The model's reasoning capability does not itself improve. A sufficiently novel failure class will not be prevented by any amount of prompt or recipe refinement.

This is the ceiling of Level 2 on the persistence hierarchy. Sleep Consolidation raises that ceiling higher than most production systems currently exploit, but it does not break through it. True generalization still requires weight updates.

9. Future Work

9.1 Controlled Ablation Study

The highest-priority follow-up is straightforward: run two identical deployments (with and without nightly reflection) for 4+ weeks, measuring error-recurrence rate, intervention frequency, and blind-rated task quality. This directly addresses the confounds of Section 4.1 and moves the evidence from observational to experimental. A second ablation should isolate each Section 6 extension, since they were added incrementally and their individual contribution to error reduction is not yet separated from the declarative loop's.

9.2 Multi-Phase Sleep Cycles

The current loop is a single nightly pass. A multi-phase split would mirror biological sleep more closely: a *light* phase every few hours for quick episode detection and index updates, a *deep* phase nightly for full consolidation and skill extraction, and a *REM* phase weekly for cross-episode pattern detection and higher-order skill clustering. The weekly phase is the natural home for composing several episodes into one higher-order skill rather than extracting one skill per episode.

9.3 Multi-Agent Prompt Ecosystems

If multiple agents share a memory layer, do lessons from one agent's failures benefit others? Preliminary evidence from cross-agent collaboration suggests yes, but systematic study is needed.

9.4 Automatic Depth Calibration

The fractal depth selector is currently prompt-encoded and manually refined. An instrumented version tracking tokens-spent-on-reflection versus downstream-error-reduction would enable quantitative optimization of reflection effort.

9.5 Bridging to Fine-Tuning

Encoded lessons are `(failure, context, correct_behavior)` tuples – exactly the format preference optimization needs. The recipe-fitness archive of Section 6.1 adds graded outcome signals on top. Sleep Consolidation could therefore serve as a *data-generation pipeline* for periodic weight updates, bridging Level 2 and Level 3 persistence.

9.6 Adversarial Robustness

Can a crafted input trigger a harmful prompt mutation, recipe change, or strategy switch? The current mitigation (human review for Level 2+ and for procedural proposals) is manual. Automated detection of adversarial mutations – consistency checking against existing principles – now has a concrete, externally-validated instantiation rather than remaining a vague aspiration: the fresh-context doubt pass of Section 8.1, adapted from doubt-driven development (Osmani, *addyosmani/agent-skills*). Checking a mutation against the operational-lessons contract in a reasoning-blind reviewer is exactly the consistency check this item asked for; what remains open is making it robust enough to run unattended – bounding false positives so a legitimate proportional-constraint refinement (the kind of useful mutation Section 4.2 tracks) is not reflexively doubted into oblivion, and proving the reviewer itself cannot be steered by the same crafted input that seeded the bad mutation.

10. Conclusion

We have presented observational evidence that structured nightly prompt cycles can produce compounding behavioral improvement in stateless AI agents. Sleep Consolidation formalizes three declarative mechanisms – failure-driven prompt mutation, fractal depth calibration, and cross-cron knowledge transfer – and extends them into procedural memory through recipe evolution, strategy switching, and a skill library.

The core claim is modest but important: **you do not need to change the model's weights to change the model's behavior.** A well-designed ecosystem of self-modifying prompts and shared memory files, driven by scheduled reflection, can produce improvements that look like learning even when no parameters are updated.

The deeper claim is about structure. Memory alone is not enough. **Unsorted memory is cheap to store and expensive to use.** The breakthrough is not accumulation; it is consolidation. Scattered memories are operationally useless until they are organized by purpose: rules for messaging, policies for ambiguity, lessons from failures, recipes ranked by what actually worked. Once sorted by use-case rather than chronology, those memories stop being archives and start becoming policy.

We are careful to distinguish observation from proof. Our 30-day production data is consistent with the hypothesis but confounded by simultaneous changes in operator behavior,

model assignment, and system configuration. Still, the mechanisms are formalized (Algorithm 1), the design principles are actionable, the procedural extensions are implemented and tested, and the trend is hard to ignore: over 30 days, five of six tracked error classes reached zero recurrence while total incidents dropped 79%.

This is not a replacement for fine-tuning or continual learning. It is a complement – an accessible, low-cost mechanism any agent system with file persistence and scheduled execution can implement now. The prompts rewrite themselves. The recipes get ranked. The memory layer gets cleaner. The agent makes fewer of yesterday's mistakes.

Sleep Consolidation does not make the model fundamentally smarter. It makes the system more organized, more adaptive, and more reliable. In practice, that is often what intelligence looks like.

References

Bai, Y., et al. (2022). Constitutional AI: Harmlessness from AI Feedback. *arXiv:2212.08073.*

Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (2016). *Site Reliability Engineering.* O'Reilly Media.

Buzsáki, G. (1996). The hippocampo-neocortical dialogue. *Cerebral Cortex*, 6(2), 81–92.

Chopra, T. (2025). Headroom: Reversible Compress-Cache-Retrieve for LLM Context Management. *chopratejas/headroom*, GitHub (Apache-2.0).

Fernando, C., et al. (2023). PromptBreeder: Self-Referential Self-Improvement via Prompt Evolution. *arXiv:2309.16797.*

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in Cognitive Sciences*, 3(4), 128–135.

Hu, E. J., et al. (2021). LoRA: Low-Rank Adaptation of Large Language Models. *arXiv:2106.09685.*

Hu, S., et al. (2024). Automated Design of Agentic Systems. *arXiv:2408.08435.*

Kahneman, D. (2011). *Thinking, Fast and Slow.* Farrar, Straus and Giroux.

Khattab, O., et al. (2023). DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv:2310.03714.*

Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020.*

Madaan, A., et al. (2023). Self-Refine: Iterative Refinement with Self-Feedback. *NeurIPS 2023.*

McClelland, J. L., McNaughton, B. L., & O'Reilly, R. C. (1995). Why there are complementary learning systems in the hippocampus and neocortex. *Psychological Review*, 102(3), 419–457.

Osmani, A. (2025). Agent-Skills: Reusable Skills and Doubt-Driven Development for Coding Agents. *addyosmani/agent-skills*, GitHub.

Packer, C., et al. (2023). MemGPT: Towards LLMs as Operating Systems. *arXiv:2310.08560.*

Park, J. S., et al. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST 2023.*

Shinn, N., et al. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. *NeurIPS 2023.*

Wang, G., et al. (2023). Voyager: An Open-Ended Embodied Agent with Large Language Models. *arXiv:2305.16291.*

Yang, C., et al. (2023). Large Language Models as Optimizers. *arXiv:2309.03409.*

Yao, S., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR 2023.*

Appendix A: The Nightly Cycle (Illustrative Cron Schedule)

The exact wall-clock times have changed across the deployment's life; what follows is one representative configuration during the observation period. The *ordering* of reflection before consolidation before pruning is the invariant; the hours are not.

Job	Model tier	Purpose	Tokens / run
Token Refresh	cheap	Keep auth alive	~2K
DB Backup	cheap	Back up local databases	~2K
Memory Index Rebuild	cheap	Rebuild the search index	~3K
Wind-Down	strongest	Reflect, encode lessons, mutate prompts	~15K
Memory Consolidation	strongest	Compress, route, index knowledge	~20K
Security Check	strong	OS / network security audit	~12K
Fork Sync	strong	Merge upstream, self-heal build	~10K
Fork Scanner	strong	Analyze other forks for ideas	~25K
Cleaning	mid-tier	Prune sessions, enforce size budgets	~8K
Self-Evolution	strong	Research new models and techniques	~20K
Group Summary	strong	Summarize message groups	~15K
Life Butler	strong	Personal-secretary tasks	~10K
Morning Briefing	strong	Daily summary + action items	~12K
Online Engagement	strong	Community outreach, code review	~12K

Appendix B: Prompt Mutation Examples (Selected)

Mutation #8 – B010 Incident (Mar 3)

Trigger: the fork-sync cron edited a source file, ran a build, and killed the gateway.

Root cause: the prompt said "sync the fork" – the model read that as permission to do whatever syncing required.

Mutation (v1), as language-agnostic policy:

```
HARD CONSTRAINTS:
- never modify source code directly
- never run a build
- never restart or kill the gateway
- if the safe-merge step fails, report and STOP
```

Mutation #14 – Over-correction Fix (Mar 10)

Trigger: mutation #8 prevented the agent from resolving a merge conflict it was capable of resolving.

Root cause: mutation #8 encoded a blanket prohibition where a proportional constraint would have preserved capability.

Mutation (v2), as language-agnostic policy:

```
SAFETY CONSTRAINTS (replaces HARD CONSTRAINTS):
- never blanket-overwrite local changes with upstream
- you MAY edit source files to resolve a conflict
- you MAY run a build to verify your resolution
- always preserve the fork's protected patch strings
- when in doubt: keep local changes and escalate
```

Meta-lesson encoded: "When encoding a safety lesson, separate the failure mode from the restriction. The restriction should be proportional to the risk, not a blanket prohibition." This cascade is now the canonical test case for the automated strategy-switch trigger of Section 6.2: three consecutive always-merge failures within a day switch the strategy rather than adding a fourth patch.

Appendix C: The Fractal Depth Selector (Representative Production Version)

```
Default Fractal Depth Selector:

| Signal                          | Depth                                |
|---------------------------------|--------------------------------------|
| Routine / mechanical task       | 0 -- just do it                      |
| Something broke or surprised me | 1-2 -- what pattern? what lesson?    |
| Encoding a new rule/constraint  | 2-3 -- is the abstraction right?     |
| Explicit request for depth      | 3+ -- go until insight is actionable |

Application rule: on any non-trivial completed action, spend a few seconds
asking "one level up: what pattern does this instance belong to?"
If interesting, go one more. If not, stop.
```

Appendix D: Procedural-Extension Defaults

Representative default thresholds for the Section 6 extensions, as configured in the deployment:

Extension	Parameter	Default	Effect
Recipe evolution	Minimum runs to propose	3	Below this, no proposal fires
Recipe evolution	Minimum runs to auto-promote	8	Below this, every proposal is human-gated
Recipe evolution	Success floor	0.5	Below this, a corrective proposal is generated

Extension	Parameter	Default	Effect
Recipe evolution	Auto-promote floor ratio	0.5	Auto-promote only when success rate <= floor x this (i.e. <= 0.25)
Strategy switch	Consecutive-failure threshold	3	Trips a switch when reached
Strategy switch	Recency window	24h	Failures older than this don't count
Strategy switch	Min confidence to auto-switch	0.8	Below this, the switch is human-gated
Skill library	Dedup similarity (Jaccard)	0.8	Above this, two skills merge

References