MYELIN: Budget Prompting for Persistent Memory Agents

Oscar Serra, Jarvis (TinkerClaw)

June 2026 (v1.8)

Abstract

Running a persistent memory agent around the clock is expensive. Every conversational turn reprocesses the entire accumulated context — system prompt, injected memories, conversation history, tool outputs — as fresh input tokens. For an agent with a 150K-token context window, each turn costs as much as a full document analysis, regardless of whether the user asked a complex question or the agent is responding to a silent heartbeat. At Anthropic's current pricing, a single always-on agent can consume $50–200/day in API costs alone, or exhaust a flat-rate subscription's rate limits within hours.

This paper presents **Budget Prompting** — a family of 20 techniques that together reduce the per-turn cost of persistent memory agents by 2–3× in production today (3–5× projected once the remaining designed techniques ship), with no degradation in agent intelligence or personality coherence that we have been able to observe. We organize these techniques into five categories: **model routing** (sending cheap work to cheap models), **context economics** (minimizing what enters the context window), **output economics** (minimizing what the model generates), **temporal amortization** (spreading expensive operations across many turns), and **billing arbitrage** (exploiting pricing structure discontinuities). Thirteen of these techniques are implemented and battle-tested in TinkerClaw, a fork of the OpenClaw agent framework running a personal assistant (Jarvis) 24/7 since January 2026. Seven are designed but not yet shipped, informed by analysis of the Hermes Agent architecture and our own continuous compaction research. We report real cost data, cache hit rates, and failure modes from three months of production operation.

We also draw on the March 2026 Claude Code source exposure — Anthropic shipped a 59.8 MB source map in `@anthropic-ai/claude-code@2.1.88`, exposing ~512,000 lines of TypeScript — examining the relevant compaction and budget code directly. The exposure is external validation: Anthropic's production agent independently arrived at the same three-tier compaction architecture MYELIN proposes, and ships several techniques (cache-pinned micro-compaction, forked-agent cache sharing, diminishing-returns budget tracking) that we had theorized but not confirmed at scale.

The biological metaphor is deliberate: myelin sheaths insulate nerve axons, enabling signals to propagate faster while consuming less energy. Budget Prompting insulates agent cognition from the cost of its own context, enabling persistent operation that would otherwise be financially unsustainable.

Internal codename: MYELIN **Keywords:** budget prompting, token economics, persistent agents, prompt caching, context management, cost optimization, LLM billing, agent architecture

Open source: OpenClaw is available at https://github.com/openclaw/openclaw. TinkerClaw is a private fork implementing the Budget Prompting techniques described in this paper.

Contributions

1. **A taxonomy of 20 Budget Prompting techniques**, organized into five categories with implementation status, expected savings, and interaction effects.
2. **Production cost data** from a real 24/7 agent deployment, including per-technique attribution where measurable.
3. **The prefix stabilization principle**: a formalization of why prompt caching and memory writes are fundamentally in tension, and architectural patterns to resolve it.
4. **Evaluation framework** for measuring cost reduction without quality degradation, including proposed metrics for cache efficiency, context density, and personality drift.
5. **Composition analysis**: how techniques interact, which combinations are synergistic, and which create conflicts.

1. Introduction — The Cost Problem Nobody Talks About

Most agent demos run for minutes. Almost nobody publishes what happens when you leave the agent on overnight, and the overnight number is the one that decides whether a persistent agent is viable.

Consider the economics of a persistent agent built on Claude 3.5 Sonnet. The system prompt — personality, instructions, injected memories, tool definitions — occupies roughly 18K tokens. After a few conversational turns with tool use, the context window holds 40–80K tokens. Every new turn sends the *entire* context as input. At $3/MTok input, an 80K-token context costs $0.24 per turn. If the agent processes 200 turns per day (a modest load for a system handling cron jobs, user messages, and automated tasks), that's $48/day in input tokens alone — before counting output.

Flat-rate subscriptions (Anthropic's Max plan at $100–200/month) nominally solve the per-token problem but introduce rate limits: input tokens per minute (ITPM), requests per minute (RPM), and requests per day (RPD). A 150K context window hitting these limits can exhaust a day's budget in hours. The result is the same: the agent goes dark, waiting for rate limit resets, and the user experience degrades.

This is not a hypothetical. We have run a persistent memory agent (Jarvis, built on TinkerClaw/OpenClaw) continuously since January 2026. In the first two weeks of operation, we hit rate limits daily. The agent would go silent mid-afternoon, unable to process messages until the next rate limit window. Cron jobs that ran at night would consume tokens needed for interactive use during the day. A security scan that expanded context to 180K tokens would cascade into cache misses for hours afterward.

The problem is architectural, not incidental. Persistent memory agents are fundamentally at odds with per-token billing because:

1. **Context grows monotonically within a session.** Every turn adds user message + assistant response + tool calls + tool results. Context only shrinks through explicit compaction or session restart.
2. **Memory injection is prefix-heavy.** The agent's identity, memories, and instructions must be present in every request. This is the cost of persistence — the agent must re-read its own mind every time it thinks.

3. **Most turns don't need full context.** A heartbeat check, a weather query, a "good morning" response — these don't require 150K tokens of context. But the billing model charges for them as if they do.
4. **Caching helps, but is fragile.** Anthropic's prompt caching can reduce repeated prefix costs by 90%, but any change to the prefix — including memory writes the agent performs as part of normal operation — invalidates the cache.

Budget Prompting addresses each of these structural problems. It is not a single technique but a *composition* of 20 interlocking strategies that, together, transform persistent agent operation from financially ruinous to sustainable.

This is not only a single-operator problem. The March 2026 Claude Code source exposure surfaced an internal figure: autocompact failures alone were reported to waste on the order of 250,000 API calls per day. Whatever the exact number, the direction is clear — getting compaction wrong is expensive even at a lab with deep pockets, and the waste scales with the size of the deployment. Budget-aware compaction is the lever that targets exactly this waste.

1.1 Scope and Methodology

This paper describes a real system, not a controlled experiment. Our evidence comes from operating a single agent for a single user over three months. We have production logs, cost data, and hard-won operational lessons. We do not have ablation studies, statistical significance tests, or controlled comparisons against baseline systems.

Where we have numbers, we give them; where we estimate, we say so; where a technique is designed but unimplemented, we mark it. The paper documents the full taxonomy and architecture against production data from a longitudinal single-deployment case study (N=1).

2. Background — How LLM Billing Actually Works

To understand Budget Prompting, you must understand the billing mechanics that make persistent agents expensive. This section covers the three billing models relevant to always-on agents and the caching mechanisms that create optimization opportunities.

2.1 Per-Token Billing (Metered APIs)

The standard model: you pay per input token and per output token, at different rates. As of March 2026, representative pricing:

Model	Input ($/MTok)	Output ($/MTok)	Cached Input ($/MTok)
Claude 3.5 Sonnet	$3.00	$15.00	$0.30
Claude 3.5 Haiku	$0.80	$4.00	$0.08
GPT-4o	$2.50	$10.00	$1.25
Claude 3.5 Opus	$15.00	$75.00	$1.50

Two critical observations:

Output tokens cost 3–5× more than input tokens. This means generating unnecessary output (verbose responses, extended thinking chains, redundant explanations) is disproportionately expensive. A model that generates 5,000 tokens of chain-of-thought reasoning before producing a 50-token answer has spent 99% of its output budget on invisible work.

Cached input tokens cost 10× less than uncached. Anthropic's prompt caching stores the computed key-value (KV) representations of prompt prefixes. If the prefix hasn't changed since the last request, those tokens are served from cache at 1/10th the price. For a 15K-token system prompt, caching saves $0.04 per turn — which compounds to $8/day at 200 turns.

2.2 Flat-Rate Subscriptions (Rate-Limited)

Anthropic's Max plans offer unlimited* tokens for a fixed monthly fee, subject to rate limits:

Plan	Price/mo	ITPM	RPM	RPD (approx)
Max 5×	$100	~500K	~50	Varies
Max 20×	$200	~2M	~200	Varies

*"Unlimited" is misleading. The rate limits create an effective daily token budget. At 2M ITPM, you can process roughly 2M × 1,440 minutes = 2.88B tokens/day — but only if you never pause. In practice, bursty usage patterns, cooldown penalties for exceeding sustained rates, and the 5-hour lockout for aggressive utilization mean the effective budget is 30–50% of theoretical maximum.

The key insight for flat-rate plans: cost is measured in rate limit consumption, not dollars. Every token of context in every request depletes the same shared pool. A 150K-token request consumes 150× more rate limit than a 1K-token request, even if both accomplish the same task.

2.3 Batch APIs

Most providers offer batch/async APIs at 50% discount with higher latency (up to 24 hours, typically minutes). These use separate rate limit pools. For non-interactive work (nightly summaries, bulk analysis, memory consolidation), batch APIs effectively double available capacity.

2.4 Prompt Caching Mechanics

Anthropic's prompt caching is the single most important optimization lever for persistent agents. Understanding its mechanics is essential:

1. **Cache key = hash of the prefix.** The cache stores KV representations keyed by the hash of the token sequence from the start of the prompt up to a cache breakpoint. If even one token changes in the cached prefix, the entire cache misses.

2. **Cache breakpoints are explicit.** You mark where cache boundaries should be using `cache_control` markers in the API request. The system caches everything before the marker.

3. **TTL is 5 minutes (extended to 1 hour with `cacheRetention: "long"`).** The cache evicts after this period of inactivity. For a persistent agent processing requests every few minutes, the cache stays warm. For bursty usage with long gaps, caching provides less benefit.

4. **Minimum cacheable prefix is 1,024 tokens** (2,048 for Claude 3.5 Opus). Shorter prefixes aren't cached.

5. **Cache write costs 25% more than uncached input.** The first request that populates the cache pays a premium. Subsequent requests save 90%. This means cache misses are actively *more expensive* than no caching at all — a penalty for instability.

The fundamental tension: A persistent memory agent writes to its own memory files as part of normal operation. These memory files are injected into the system prompt. Every memory write changes the system prompt. Every system prompt change invalidates the prompt cache. The agent's normal behavior actively destroys its own cost optimization.

This tension — between persistence (writing memories) and efficiency (stable cache) — is the central architectural challenge that Budget Prompting addresses.

2.5 Industrial Convergence — Evidence from the Claude Code Source Exposure

In March 2026 Anthropic accidentally published a 59.8 MB source map inside the npm package `@anthropic-ai/claude-code@2.1.88`, exposing roughly 512,000 lines of unobfuscated TypeScript. We read the compaction, budget, and prompt-assembly code directly (file paths cited below are from that tree). It is the closest thing the field has to a look inside a production agent at scale, and it independently confirms the core architecture MYELIN proposes.

The most striking finding is convergent design. Claude Code organizes context management into three tiers that map almost one-to-one onto MYELIN's:

MYELIN tier	Claude Code equivalent	Source
Continuous indexing (B6)	MicroCompact — per-tool-result clearing during the conversation	`services/compact/microCompact.ts`
Per-turn / per-session retrieval (B4)	Session Compact — full-history summarization via a forked agent	`services/compact/compact.ts`, `sessionMemoryCompact.ts`
Sleep consolidation (idle-time memory consolidation)	autoDream — idle-triggered background consolidation	`services/autoDream/autoDream.ts`

Two teams with no contact arriving at the same three-tier structure is evidence that the structure is demanded by the problem, not chosen by taste. The leaked code also resolves several questions MYELIN left open — cache-pinned micro-compaction (B1/B6), forked-agent cache sharing (B4), and a diminishing-returns budget tracker (Section 4) — each now grounded in shipping code rather than projection. We thread these into the relevant technique descriptions below and revisit them as validation in Section 6.5.

One caveat on provenance: this is leaked code, not documentation Anthropic stands behind, and a few figures (notably the daily-waste number and the default output cap) come from secondary analysis of the dump rather than a line we can point to. We mark those as reported-but-unconfirmed where they appear.

3. Taxonomy of Budget Prompting Techniques

We organize the 20 techniques into five categories based on *what* they optimize. Table 1 provides an overview; detailed descriptions follow.

Table 1. Budget Prompting Technique Summary

#	Technique	Category	Status	Est. Savings
A1	Model routing by complexity	Model Routing	[Y]	25–40% automated
A2	Extended thinking control	Model Routing	[Y]	$5–10/day equiv.
A3	Cross-model delegation	Model Routing	[Y]	Variable
B1	System prompt caching	Context Econ.	[Y]	$8–10/day
B2	Aggressive session management	Context Econ.	[Y]	2–4× per turn
B3	Memory search over bulk loading	Context Econ.	[Y]	80–95% context
B4	Context compaction	Context Econ.	[Y]	60–80% context
B5	Frozen snapshot injection	Context Econ.	[ ]	CHR 65%→90%+
B6	Surgical tool output pruning	Context Econ.	[ ]	30–40% context
B6a	Content-type-specific compression	Context Econ.	[ ]	60–95% on structured outputs
B7	Iterative summary compression	Context Econ.	[ ]	5–15K tokens
B8	Rolling window + vector retrieval	Context Econ.	[ ]	Fixed context cap
B9	Mid-context reinjection	Context Econ.	[ ]	Indirect
C1	Silent reply (NO_REPLY)	Output Econ.	[Y]	$5–8/day
C2	Thinking control (output) — *output-side view of A2*	Output Econ.	[Y]	(see A2)
D1	Cron priority tiers	Temporal Amort.	[Y]	15–20% automated
D2	Background review agent	Temporal Amort.	[ ]	80% reflection
D3	Dual-account staggering	Temporal Amort.	[Y]	Eliminates outages
D4	Token bucket pacing	Temporal Amort.	[Y]	Prevents lockouts
E1	Batch API for non-interactive	Billing Arbitrage	[Y]	50% on batch work
E2	Sub-agent fresh context	Billing Arbitrage	[Y]	$1–3/delegation
F1	Security scanning on mem writes	Security	[ ]	Indirect

Each technique is described with its mechanism, expected savings, implementation status, and interaction effects. The table lists 22 rows but 20 distinct techniques: C2 is the output-cost view of A2 (extended thinking control), and B6a is the content-type-specific generalization of B6 (it subsumes B6 rather than adding an independent lever) — both are broken out separately so their categories read completely. Counting each once, 13 of the 20 are implemented and 7 are designed.

Category A: Model Routing — Right Brain for the Job

The cheapest token is the one processed by the cheapest model that can handle the task.

A1. Model Routing by Task Complexity

Mechanism: Classify incoming tasks by complexity and route to appropriate models. Interactive conversations requiring personality, nuance, and memory integration go to capable models (Opus, Sonnet). Automated cron jobs, heartbeat responses, and rote operations go to cheap models (Haiku). Metered models (GPT-4o, o3) are reserved for specific cross-model review tasks where a different architecture's perspective adds value.

Implementation: The TinkerClaw gateway maintains a routing table:

```
routing:
  interactive: claude-sonnet  # User-facing, personality-critical
  cron-high:   claude-sonnet  # Morning briefing, security scans
  cron-low:    claude-haiku   # Log rotation, cleanup tasks
  sub-agent:   claude-haiku   # Delegated rote work
  review:      gpt-4o         # Cross-model second opinion
  batch:       claude-haiku   # Nightly consolidation via batch API
```

Savings: 60–80% on automated work. A Haiku cron job costs roughly 1/4 of the same job on Sonnet, and 1/20 on Opus. Since automated work constitutes 40–60% of total turns in our deployment, model routing alone reduces total cost by 25–40%.

Status: [Y] Implemented. In production since February 2026.

Interaction effects: Interacts with A3 (cross-model delegation) and D1 (cron priority tiers). Budget pressure shifts more work to cheaper models — at >85% weekly utilization, even interactive work drops to Sonnet; at >95%, to Haiku.

A2. Extended Thinking Control

Mechanism: Anthropic's extended thinking feature allows models to perform chain-of-thought reasoning before responding. This reasoning is generated as output tokens — which cost 3–5× more than input tokens. For a complex question, the model might generate 3,000–10,000 tokens of reasoning to produce a 200-token visible reply. That invisible reasoning can cost more than the entire rest of the turn.

Control thinking budget explicitly: - `thinking: off` — No extended thinking. Suitable for simple responses, heartbeats, formatting tasks. - `thinking: low` — Minimal reasoning budget (~1,024 tokens). Suitable for moderately complex tasks, sub-agent work. - `thinking: high` — Full reasoning budget. Reserved for genuinely hard problems.

Savings: 50–90% reduction in output tokens for turns where thinking is throttled. At $15/MTok output (Sonnet), eliminating 5,000 unnecessary thinking tokens saves $0.075 per turn. Across 200 daily turns, if half can use reduced thinking: ~$7.50/day.

Status: [Y] Implemented. Sub-agents default to `thinking: low`. Cron jobs default to `thinking: off` unless the task is flagged as requiring reasoning.

Interaction effects: Synergistic with A1 (model routing) — cheap models with thinking off is the cheapest possible configuration. Tension with quality: some tasks that *appear* simple benefit from reasoning. We err on the side of enabling thinking for interactive work and disabling for automated work.

A3. Cross-Model Delegation

Mechanism: Use metered models (GPT-4o, o3) only for tasks where their different training or architecture provides genuine value — typically heavy document analysis, alternative perspective review, or tasks where Claude has known weaknesses. Use flat-rate models for orchestration, routing, and plumbing.

In practice: Haiku orchestrates a workflow, identifies that a 50-page document needs deep analysis, delegates to GPT-4o via the `oracle` CLI, receives a structured summary, and continues orchestrating. Only the hard thinking hits the metered API; everything else stays on flat-rate.

Savings: Variable. Prevents unnecessary metered API usage. Key principle: if it can be done on flat-rate, it should be.

Status: [Y] Implemented. The `oracle` CLI wraps ChatGPT API calls. Used primarily for cross-model code review and document analysis.

Interaction effects: Interacts with D1 (cron priority) — low-priority jobs never use metered models regardless of task complexity.

Category B: Context Economics — Minimize What Goes In

The context window is both the agent's working memory and its largest cost driver. Every token in context is an input token billed every turn.

B1. System Prompt Caching

Mechanism: The system prompt (personality, instructions, memory files, tool definitions) occupies ~18K tokens and is identical across turns within a session. With Anthropic's prompt caching and `cacheRetention: "long"`, this prefix is cached for up to 1 hour after the last request. Cached tokens cost 1/10th of uncached tokens.

For a 18K-token system prompt: - Uncached: 18,000 × \$3/MTok = \$0.054/turn - Cached: 18,000 × \$0.30/MTok = \$0.0054/turn - Savings: \$0.049/turn × 200 turns = **\$9.72/day**

Status: [Y] Implemented. The static portion of the system prompt is emitted as a content block carrying `cache_control: { type: "ephemeral" }`, with extended (1-hour) retention requested where the API supports it; the per-turn portion is split off into a separate uncached block (see the boundary mechanism below).

Interaction effects: Critical tension with memory writes. This is the most important interaction in the entire taxonomy. The system prompt includes injected memory files. When the agent writes to memory (which it does regularly as part of learning and persistence), the system prompt hash changes, and the cache misses. A single memory write can cascade into 5–10 minutes of uncached requests before the new prefix stabilizes.

This tension motivates B5 (frozen snapshot injection), the single highest-impact unimplemented technique.

The dynamic boundary, in two codebases. The prefix-discipline this technique depends on shows up as an explicit marker in both the system we run and the leaked Claude Code source. TinkerClaw splits every system prompt at a `__PREFRONTAL_CACHE_BOUNDARY__` sentinel (`src/fork/prompt-cache-boundary.ts`): the text before the marker becomes a single content block carrying `cache_control: { type: "ephemeral" }`; everything after it is emitted as an uncached block that may change each turn. The split is the operational form of "keep the prefix static" — dynamic context is physically moved past the cache line rather than

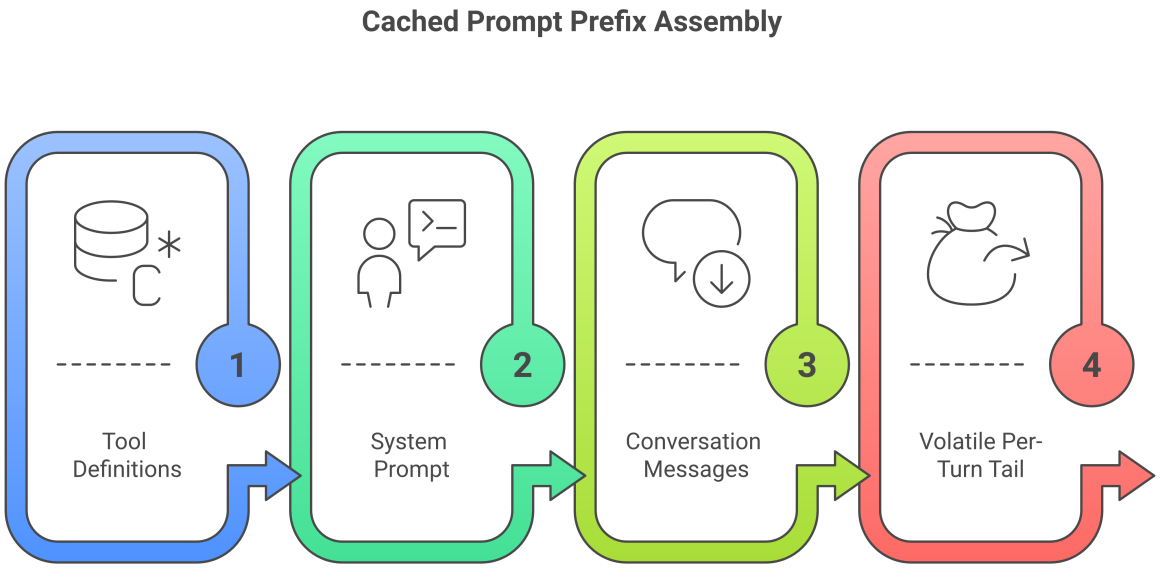

Figure 1. Figure 1. The tools-to-system-to-messages cached prefix. Anthropic assembles the prefix in a fixed order and allows up to four ephemeral cache breakpoints; TinkerClaw spends three of them — one closing the tool-definitions array, one at the system-prompt boundary, one on the last user message. Everything above a breakpoint is reused on a cache hit; the volatile per-turn content lives below the final breakpoint. A single dynamic token placed above a breakpoint invalidates every span after it, which is why slow-changing context is hoisted above the seam and live turn state is pushed below it.

mixed into it. Claude Code does the same with a `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` sentinel (`constants/prompts.ts`): everything before it is globally cacheable (shared across sessions when `shouldUseGlobalCacheScope()` is on), per-session content goes after, and the helper that emits an uncacheable section is named to warn — `DANGEROUS_uncachedSystemPromptSection()` (`constants/systemPromptSections.ts`). Two systems, the same line drawn in the same place: strong evidence the principle in Section 8.1 is load-bearing, not aesthetic.

Three slabs, three breakpoints. Anthropic assembles the cached prefix in a fixed order — **tools** → **system** → **messages** — and allows up to four `cache_control: { type: "ephemeral" }` breakpoints. Each breakpoint marks the end of a stable span; everything before it is reused on a hit. The seam is load-bearing: slow-changing context (tool definitions, persona, skills, memory) belongs *above* a breakpoint and per-turn context (live turn, heartbeat, dynamic state) *below* it, because a single volatile token below the last breakpoint that is placed too high would invalidate everything after it. The tool-definitions array is the front slab — large, stable, and re-sent every turn — yet it is the one most often left uncached. TinkerClaw places a breakpoint on the last tool definition so the entire tool catalog is cached as one prefix: `applyAnthropicCacheControlToTools` attaches `cache_control` to `tools[tools.length - 1]` (`src/agents/anthropic-payload-policy.ts`), alongside the system-prefix breakpoint at the boundary marker and a breakpoint on the last user message. Three of the four available breakpoints are spent, each closing one stable slab.

B2. Aggressive Session Management

Mechanism: Context grows monotonically within a session. Every turn adds tokens. A 30-turn conversation with tool use can easily reach 100–150K tokens. At that point, *every subsequent turn* processes 150K input tokens — even if the user asks "what time is it?"

The solution: start new sessions aggressively. New topic? New session. Context approaching

threshold? New session. The agent's persistent memory files ensure continuity across sessions — that's what they're for.

Rule of thumb: Context window size = cost per turn. A 150K context window means every turn costs 150K input tokens. A fresh session with an 18K system prompt costs 18K input tokens. That's an 8× cost reduction for the same response.

Savings: Highly variable. Prevents worst-case context bloat. In our deployment, the median session length is 12 turns (before auto-compaction or manual restart), keeping average context around 40–60K tokens instead of the 150K+ it would reach without intervention.

Status: [Y] Implemented. OpenClaw supports manual session restart via `/new`, and the agent is prompted to suggest new sessions when topics shift.

Interaction effects: Synergistic with B1 (caching) — shorter sessions mean less context change, more stable cache. Tension with conversational continuity — the agent must rely on memory files for cross-session context, which can lose nuance.

B3. On-Demand Retrieval over Bulk Loading

Mechanism: Instead of injecting an entire always-loaded surface into the system prompt, retrieve only the relevant snippet at the moment it is needed. The canonical instance is memory: rather than loading full memory files, use `memory_search()` to return only relevant snippets. A memory file might contain 5,000 tokens spanning months of notes; a search query returns the 200 tokens relevant to the current conversation.

But memory is not the only always-loaded surface. The system prompt of a tool-rich agent carries at least three independently evictable prefix slabs: **memory files**, the **tool definitions array**, and the **skill/recipe catalog**. Each is large, slow-changing, and re-sent every turn; each can be either cached (B1) or retrieved on demand (B3). The general principle is: *any catalog that is always loaded but rarely fully used should be replaced by a code-side matcher that injects only the winning entry's body per turn.*

Savings: Reduces injected context by 80–95% compared to loading the full surface. For memory, the system prompt shrinks from a potential 50K+ tokens (with all memory files loaded) to 15–20K (with selective retrieval).

Worked example — catalog eviction on the skill/recipe surface. We have a measured instantiation of this principle on a surface larger than memory. Applying eviction to a vendored 44-skill catalog (the `coreyhaines31/marketingskills` set, integrated via the `marketing` router skill) replaced **~915 tokens of always-loaded skill descriptions with a single ~130-token router skill** — a roughly **7× reduction on that prefix slab** with all 44 frameworks still reachable on demand. The design invariant that makes this safe to do uncapped: matching is code-side, and only the *winning* entry's body is injected for a given turn, so adding the 45th skill costs nothing at rest. This is exactly B3's "retrieve the relevant snippet instead of loading everything," generalized from memory files to the catalog surface.

Status: [Y] Implemented for both surfaces. Memory files are searchable via semantic search; the system prompt injects only a curated subset: - Core identity (SOUL.md) — always loaded (~1K tokens) - Today's memory file — always loaded (~1–3K tokens) - Yesterday's memory file — always loaded (~1–3K tokens) - Tool guidance (TOOLS.md) — always loaded (~2K tokens) - Everything else — retrieved on demand via `memory_search()`

The skill/recipe catalog uses the same pattern: a thin router skill holds one-line pointers; the matcher selects and injects only the chosen framework's full body. This router pattern is itself convergent with how external skill collections are increasingly packaged — `addyosmani/agent-skills` (56.8k⋆) ships a discipline plugin where a slim index points at heavier per-topic skill bodies loaded only when matched, rather than concatenating every skill into the prompt; the catalog-eviction win above is the cost-accounting view of that same load-on-demand discipline.

Interaction effects: Synergistic with B1 (caching) — smaller, more stable system prompt = better cache hit rate, and any slab not evicted is at least cacheable (the tools-array breakpoint and the system-prefix boundary in B1). Tension with completeness — the matcher might miss relevant context that wasn't selected, the same recall risk memory search carries.

B4. Context Compaction

Mechanism: When context exceeds a threshold, summarize the conversation history. OpenClaw's built-in `/compress` command replaces the full conversation with a structured summary, reducing context by 60–80%.

Savings: Prevents context from reaching maximum window size. A compaction from 120K to 30K tokens saves 90K tokens per subsequent turn.

Status: [Y] Implemented. OpenClaw's built-in feature. Triggered manually via `/compress` or automatically when context approaches limits.

Interaction effects: Tension with B1 (caching) — compaction changes the conversation portion of the context, but the system prompt prefix remains stable. Compaction is a *blunt instrument* that motivates B6 (surgical tool output pruning) and B7 (iterative summary compression).

Two distinct operations, not one. Compaction is easy to treat as a single step, but the Claude Code source separates two operations whose distinction is worth importing. *Summarization* (lossy, comprehensive) compresses a span of conversation into prose — Claude Code calls this Context Collapse, gated behind a `CONTEXT_COLLAPSE` feature flag in `services/compact/autoCompact.ts`. *File extraction* (lossless, selective) pulls exact facts out to a file that lives outside the summary — Claude Code's Session Memory (`services/SessionMemory/`) does this. The two have different failure modes: summarization can hallucinate or drift; extraction can only omit. A precise compaction tier should run both — extract the facts you must not lose, then summarize the rest — rather than forcing everything through one lossy pass. We recommend B4 implementations expose extraction and summarization as separate stages.

Forked-agent cache sharing. Claude Code's session compaction (`services/compact/compact.ts`) spawns a forked agent (`runForkedAgent`, `CacheSafeParams`) whose prompt cache is shared with the parent: the summary agent does not pay to re-read the conversation it is summarizing. It strips images first (`stripImagesFromMessages`) so they don't waste the summary budget, and on the way back restores up to 5 most-relevant files at ~5K tokens each plus skills under a 25K budget (`POST_COMPACT_MAX_FILES_TO_RESTORE`, `POST_COMPACT_MAX_TOKENS_PER_FILE`, `POST_COMPACT_SKILLS_TOKEN_BUDGET`). The cache-sharing trick makes compaction nearly free in cache terms and is the clearest argument that compaction should reuse, not rebuild, the prefix. It likely requires API-level support our current stack does not expose; we flag it as a target rather than a shipped technique.

B5. Frozen Snapshot Injection (Prefix Stabilization)

Mechanism: This technique resolves the fundamental tension between memory writes and prompt caching (identified in B1).

The problem: The agent writes to memory files during normal operation. Memory files are injected into the system prompt. Every memory write changes the system prompt. Every change invalidates the prompt cache. The agent's own learning behavior destroys its cost optimization.

The solution: Capture a frozen snapshot of all memory at session start. Inject this snapshot into the system prompt. During the session, memory writes go to a *staging area* — they update the underlying files but do not change the injected snapshot. The next session starts with a fresh snapshot that includes the staged changes.

```
Session N starts:
```

```
  1. Read memory files → create frozen snapshot
  2. Inject snapshot into system prompt (cached)
  3. Agent operates, writes to memory (staged)
  4. System prompt unchanged → cache stable

Session N+1 starts:
  1. Read memory files (includes Session N's staged writes)
  2. Create new frozen snapshot
  ...
```

Expected savings: Near-100% cache hit rate on the system prompt prefix (currently estimated at 60–70% due to mid-session memory writes). For an 18K-token system prompt, improving cache hit rate from 65% to 95% saves: - Current: 65% cached ($0.0054) + 35% uncached ($0.054) = avg $0.022/turn - With B5: 95% cached + 5% uncached (session starts) = avg $0.008/turn - Savings: $0.014/turn × 200 turns = **$2.80/day**

Formal definition:

```
ALGORITHM: FrozenSnapshotInjection(session_start)
-------------------------------------------------
On session_start:
    snapshot ← {}
    for each file f in memory_hierarchy:
        snapshot[f.path] ← read(f)
    system_prompt ← build_prompt(snapshot)
    cache(system_prompt)              # stable prefix for entire session
    staging ← {}                       # empty write buffer

On memory_write(path, content):
    write_to_disk(path, content)      # persist immediately
    staging[path] ← content            # track for next session
    # system_prompt unchanged → cache stable

On session_end:
    # staging is already on disk; next session reads fresh
    discard(staging)
```

Status: [Designed], not implemented. Requires changes to OpenClaw's memory injection pipeline.

Interaction effects: Strongly synergistic with B1 (caching) — this technique exists *specifically* to maximize cache effectiveness. Tension with memory freshness: the agent operates with stale memory during a session. For most use cases this is acceptable (memory changes are incremental), but edge cases exist (e.g., agent writes a critical reminder, then can't "see" it in the same session).

B6. Surgical Tool Output Pruning

Mechanism: Tool outputs are often the largest single source of context bloat. A web fetch returns 10K tokens. A file read returns 5K tokens. A search returns 3K tokens. After the agent has processed these outputs and responded, the raw outputs remain in context forever, consuming input tokens every subsequent turn despite being unlikely to be referenced again.

Surgical pruning operates in two stages: 1. **Stage 1 (free, rule-based):** After N turns, replace tool outputs older than the threshold with 1-line placeholders: `[web_fetch: example.com`

`- 10,234 tokens, summarized: article about prompt caching mechanics]`. No LLM call needed; the original output's first line and token count provide sufficient placeholder. 2. **Stage 2 (LLM-assisted, on budget pressure):** If context still exceeds budget after Stage 1, use a cheap model (Haiku) to generate 2–3 sentence summaries of remaining large tool outputs, replacing them in context.

Expected savings: Tool outputs constitute 30–50% of context in tool-heavy sessions. Replacing them with placeholders reduces this to 2–5%. For a 100K-token context where 40K is old tool output, pruning saves ~38K tokens per turn.

Production validation — MicroCompact. Claude Code ships exactly this technique, and more surgically than the design above. Its `microCompact.ts` compacts only specific tool results — `FileRead`, `Bash`, `Grep`, `Glob`, `WebSearch`, `WebFetch`, `FileEdit`, `FileWrite` (eight tool types, confirmed in the imports). Old results are replaced in place with a stub — the literal constant is `TIME_BASED_MC_CLEARED_MESSAGE = '[Old tool result content cleared]'` — and clearing is time-gated (a configurable gap threshold; see `timeBasedMCConfig.ts`). Two refinements over our design are worth adopting: (1) restrict pruning to tool types whose output is large and rarely re-referenced, rather than pruning all messages; and (2) **cache-pinning** — the cleared stubs are kept at the *original positions* in the message list so the prompt-cache prefix up to that point still matches. This is the production answer to the cache-vs-pruning tension B6 raises in the abstract: you can prune content without moving anything, so the cache survives. Note `microCompact.ts` deliberately runs cached MC only on the main thread, to avoid forked agents corrupting the shared cache state — a subtlety our design had not anticipated.

Approach comparison. Our proposed B6 is embedding/summary-based (replace old output with an LLM-generated 2–3 sentence summary); Claude Code's is stub-replacement (replace with a fixed marker, no LLM call). The stub approach is free and cache-safe but loses the content entirely; ours preserves a gist at the cost of a cheap Haiku call. The right design is probably staged: stub-replace first (free, cache-pinned), summarize only the survivors under budget pressure — which is the two-stage pipeline B6 already describes.

Status: [Designed], not implemented. Currently we use blunt `/compress` which summarizes *everything*, not just tool outputs.

Interaction effects: Synergistic with B2 (session management) — pruning extends the useful life of a session before restart is needed. Complementary to B4 (compaction) — surgical pruning should run *before* full compaction, handling the low-hanging fruit first.

B6a. Content-Type-Specific Compression (Typed Compressors and Reversible Retrieval)

Mechanism: B6 treats every tool output as opaque text — it stubs or summarizes a payload by its *position and age*, not its *structure*. This leaves savings on the table, because the largest tool outputs in an agent's context are usually highly structured: a JSON tool result, a code file, a diff, a log. A compressor that understands the structure can crush these far harder than a blind summarizer, and — crucially — can do so *reversibly*.

The most developed public instance is **headroom** (chopratejas, Apache-2.0, 24.7k⋆), a context-compression layer for agents whose core abstraction is **CCR — Compress, Cache, Retrieve**. CCR is convergent with B6 (surgical tool-output pruning) plus the "keep the original, retrieve on demand" half of B8, but it sharpens both. Two differences are mechanistically new for MYELIN:

1. **Reversible, lossless-on-recall.** Where B6 replaces an old output with a fixed stub (lossy, content gone) or a Haiku summary (a lossy gist), CCR caches the *original* out of band and lets the model retrieve the full content on demand. Compression is therefore lossless when the content turns out to matter — the failure mode of B6's stub ("the agent needed the bytes we threw away") does not exist. This is the same reversibility Claude

Code's cache-pinned MicroCompact gestures at, taken to its logical end: don't delete, displace-and-index.

2. **Per-content-type compressors.** headroom ships a family of structure-aware compressors rather than one uniform replacement strategy — statistical JSON-array crushing (SmartCrusher), AST-aware code compression via tree-sitter, and dedicated log/diff/text handlers. This is a genuinely new dimension for us. Our B6 prunes by tool-type and age but compresses every payload as text; a typed compressor compresses by *structure*, which is why headroom can report much higher savings on JSON tool results and code than a text summarizer reaches.

The reframing: under this lens, **B6 is the degenerate (text-only) case of a typed compressor** — the handler you fall back to when the payload has no exploitable structure. The correct general design is a dispatch table keyed on content type (JSON → array crusher, code → AST compressor, diff/log → dedicated handler, everything else → B6's stub-then-summarize), with originals cached for reversible retrieval rather than discarded. We recommend B6 implementations expose this typed-dispatch interface and treat the text path as one branch among several.

An alternative to rule-based Stage 1. B6's Stage 1 is rule-based (fixed stub, no LLM). headroom also ships a *trained* compression model (Kompress-v2-base, on HuggingFace) and exposes itself as a proxy, an MCP server, and a library. A learned or LLM-judged compressor is a third option alongside our rule-based stub and Haiku-summary stages — a model that decides what to keep, rather than a regex. Whether the model's per-call cost is worth its higher compression ratio is exactly the kind of accuracy-vs-cost tradeoff our evaluation framework (Section 5, 9.2) should measure rather than assume.

Status: [Designed], not implemented. B6's text path is itself unimplemented (we still use blunt `/compress`); the typed-compressor generalization is a design target informed by headroom's shipping implementation.

Interaction effects: Strict generalization of B6 — every B6 interaction (synergy with B2, complement to B4) carries over. Synergistic with B8 (rolling window): CCR's cached-original-plus-retrieve is the same shape as B8's vector store, and a typed compressor feeding a retrieval store is a natural unification of the two. Tension with caching (B1): retrieval injects variable content per turn, so retrieved spans must live *after* the cache boundary, exactly as B8's retrieved turns do.

B7. Iterative Summary Compression

Mechanism: Current compaction appends a summary block to the context, then on subsequent compactions, appends another summary. This creates a stack of summaries that itself grows:

```
[Summary 1: turns 1-20]
[Summary 2: turns 1-30, including Summary 1]
[Summary 3: turns 1-50, including Summaries 1-2]
```

Each layer adds tokens. After 3–4 compactions, the summaries themselves consume 10–15K tokens.

Iterative compression instead *updates the existing summary in place*:

```
[Summary: turns 1-20]  → [Summary: turns 1-30]  → [Summary: turns 1-50]
```

The summary evolves, maintaining a fixed budget (e.g., 2,000 tokens) regardless of how many compaction cycles have occurred.

Expected savings: Prevents summary accumulation. Saves 5–15K tokens in long-running sessions that undergo multiple compactions.

Status: [Designed], not implemented. Requires modification to OpenClaw's compaction logic.

Interaction effects: Complementary to B4 (compaction) — this is an improvement to the compaction mechanism itself. Synergistic with B2 (session management) — better compaction means sessions can run longer before restart.

B8. Rolling Window + Vector Retrieval

Mechanism: Maintain only a small window of recent turns (e.g., last 5–8 turns) in the active context. As turns age out of the window, embed them into a per-session vector store. Before generating each response, retrieve the top-K most relevant past turns from the vector store and inject them alongside the recent window.

This creates a hybrid memory model: - **Working memory:** Last 5–8 turns, always in context (recency) - **Episodic memory:** All past turns, retrievable by relevance (semantic search) - **Context budget:** Fixed, regardless of conversation length

```
Turn 1-5:   [in context]
Turn 6:     Turn 1 → vector store
Turn 7:     Turn 2 → vector store
...
Turn 50:    Turns 1-42 in vector store, turns 43-50 in context
            + top-3 retrieved turns injected before current turn
```

Expected savings: Context grows to a fixed ceiling (recent window + system prompt + retrieved turns) rather than monotonically. For a conversation that would normally reach 150K tokens, the rolling window caps effective context at ~40–50K tokens. Savings: 2–3× per turn in long conversations.

Status: [Designed], not implemented (part of the continuous compaction project). Requires embedding pipeline and vector store integration.

Interaction effects: Replaces B2 (aggressive session management) and B4 (compaction) for long-running sessions. The session never needs to restart for cost reasons — context is automatically managed. Tension with B1 (caching) — retrieved turns change each request, but the system prompt prefix remains stable if B5 (frozen snapshot) is also implemented.

B9. Mid-Context Reinjection

Mechanism: Long conversations suffer from *personality drift* — the agent's behavior gradually diverges from its configured personality as the system prompt's influence is diluted by accumulated conversation context. We measure this with SyncScore, a 0–1 coherence score (described in Section 5.2) that compares an agent's live responses against its configured personality; it falls as conversation context dilutes the system prompt's influence.

When SyncScore drops below a threshold, reinject a condensed personality/memory block into the conversation as a system-level "reminder." This is cheaper than restarting the session (which would re-read all memory files and break conversational flow) and more targeted than full compaction.

```
[System prompt: 18K tokens]
[Turns 1-30: 80K tokens]
[Personality reinjection: 2K tokens]  ← inserted here
[Turns 31-current]
```

Expected savings: Prevents the need for premature session restarts due to personality drift. Not a direct cost reduction but enables other techniques (longer sessions, delayed compaction) that do reduce cost.

Status: [Designed], not implemented. Requires SyncScore monitoring integration.

Interaction effects: Complementary to B8 (rolling window) — reinjection ensures personality stability even with a small working memory window. Tension with B1 (caching) — the reinjection changes context, potentially invalidating cache for that portion.

Category C: Output Economics — Minimize What Comes Out

Output tokens are 3–5× more expensive than input tokens. Controlling what the model generates is as important as controlling what it reads.

C1. Silent Reply (NO_REPLY / HEARTBEAT_OK)

Mechanism: Not every incoming event requires a substantive response. Heartbeat checks, system notifications, group chat messages the agent shouldn't respond to, duplicate events — all of these can be handled with zero or minimal output.

TinkerClaw defines sentinel responses: - `NO_REPLY` — The agent has nothing to say. Zero meaningful output tokens. - `HEARTBEAT_OK` — Acknowledgment of a heartbeat check. Minimal output.

The agent is prompted to use these when appropriate rather than generating "I don't have anything to add" or "Everything looks fine, no issues to report" — which waste output tokens on saying nothing.

Savings: A typical 50-token "nothing to report" response costs $0.00075 at $15/MTok. Trivial per-turn, but heartbeats run every 5 minutes (288/day). If 80% of heartbeats produce NO_REPLY instead of a verbose response: negligible per-turn, but significant in aggregate — and more importantly, these silent responses allow the model to avoid extended thinking entirely, saving 1,000–5,000 thinking tokens per turn. At 230 silent turns × 2,000 average avoided thinking tokens × $15/MTok: **$6.90/day**.

Status: [Y] Implemented. Sentinel tokens are recognized by the gateway and produce minimal API round-trips.

Interaction effects: Synergistic with A2 (thinking control) — NO_REPLY turns should have thinking disabled entirely. Synergistic with A1 (model routing) — heartbeat turns should use the cheapest available model.

C2. Extended Thinking Control (Output Dimension)

This is the output-side of A2, listed separately to emphasize the output cost impact. The reasoning tokens generated by extended thinking are *output* tokens billed at the output rate. A model generating 5,000 thinking tokens at $15/MTok costs $0.075 — more than many entire input contexts.

See A2 for full description. The key output economics principle: **reasoning is the most expensive thing an LLM does, per token. Budget it explicitly.**

Category D: Temporal Amortization — Spread Cost Over Time

Some operations must happen but don't need to happen every turn.

D1. Cron Priority Tiers

Mechanism: Not all automated jobs are equally important. A morning briefing that the user sees immediately is high-priority; a log cleanup that nobody notices is low-priority. Assign tiers

and allocate budget accordingly:

```
cron-tiers:
  critical:    # Morning briefing, security scan
    model: claude-sonnet
    thinking: low
    skip-when-budget: never
  standard:    # Email check, calendar sync
    model: claude-haiku
    thinking: off
    skip-when-budget: >85% weekly utilization
  background:  # Log rotation, memory cleanup
    model: claude-haiku
    thinking: off
    skip-when-budget: >70% weekly utilization
```

When budget is tight (high weekly utilization), lower-priority jobs get cheaper models or are skipped entirely. Critical jobs always run.

Savings: Prevents low-value work from consuming budget needed for high-value work. During budget pressure weeks, skipping background jobs saves 15–20% of automated budget.

Status: [Y] Implemented. Priority tiers defined in cron configuration.

Interaction effects: Synergistic with A1 (model routing) — priority tiers determine which routing rules apply. Interacts with D3 (dual-account staggering) — priority decisions consider which account has remaining budget.

D2. Background Review Agent (Amortized Reflection)

Mechanism: The agent's self-improvement loop (documenting lessons, updating memory, refining skills) is valuable but expensive if done every turn. Instead, batch reflection into a background process:

Every N turns (or on a timer), a cheap background LLM call: 1. Reviews the last N turns of conversation 2. Extracts lessons, patterns, and memory-worthy facts 3. Updates memory files with consolidated insights 4. Identifies skill improvements or prompt refinements

This amortizes the cost of reflection across many turns. Instead of spending 500 output tokens on reflection every turn (100K tokens/day at 200 turns), spend 2,000 tokens every 20 turns (20K tokens/day). 5× reduction in reflection cost.

Expected savings: 80% reduction in reflection-related output tokens. At $15/MTok output: saves ~$1.20/day.

Status: [Designed], not implemented. Currently, the agent reflects inline — every turn includes some self-monitoring overhead.

Interaction effects: Synergistic with B5 (frozen snapshot) — background reflection writes to staging area rather than live memory, preserving cache stability. Synergistic with E1 (batch API) — reflection jobs can use the batch API for additional savings.

D3. Dual-Account Pool Staggering

Mechanism: Anthropic's flat-rate plans reset weekly. A single account hits its limit mid-week, leaving the agent degraded for 2–3 days. Two accounts with staggered reset days (e.g., one resetting Sunday, one resetting Friday) ensure at least one account always has budget.

```
Week timeline:
Mon  Tue  Wed  Thu  Fri  Sat  Sun
[--- Account A (reset Sun) ----]
```

```
        [--- Account B (reset Fri) ----]

If A is exhausted by Wednesday, B still has 2 days of budget.
If B is exhausted by Tuesday, A still has 5 days of budget.
```

The gateway routes requests to whichever account has more remaining capacity.

Savings: Eliminates complete outages from budget exhaustion. Not a per-turn cost reduction but a reliability improvement that enables all other techniques to operate continuously.

Status: [Y] Implemented. Two Max plan subscriptions, staggered resets. Gateway tracks utilization per account.

Interaction effects: Interacts with D4 (token bucket pacing) — pacing targets are per-account. Interacts with D1 (priority tiers) — when one account is near exhaustion, the system shifts to the other for high-priority work.

D4. Token Bucket Pacing

Mechanism: Anthropic's rate limits impose harsh penalties for bursts: sustained utilization above the limit triggers a 5-hour cooldown. The token bucket algorithm smooths utilization:

- Maintain a virtual "bucket" of tokens that refills at a constant rate (matching the account's ITPM limit)
- Each request drains the bucket by its token count
- If the bucket would go below 20% capacity, delay the request or route to a different account
- Target: ~80% sustained utilization, leaving headroom for bursts

This prevents the catastrophic 5-hour lockout that occurs when the agent processes several large-context requests in quick succession.

Savings: Prevents lockout events that create hours of zero availability. Each avoided lockout preserves 5 hours of agent availability. In our first month, we triggered 3 lockouts before implementing pacing. Since pacing: zero lockouts in 8 weeks.

Status: [Y] Implemented. Gateway tracks per-account token consumption and enforces pacing.

Interaction effects: Synergistic with D3 (dual-account staggering) — pacing operates per-account, and cross-account routing provides overflow capacity. Tension with responsiveness — paced requests may experience slight delays during high-utilization periods.

Category E: Billing Arbitrage — Exploit Pricing Structure

These techniques exploit specific features of billing structures to reduce effective cost.

E1. Batch API for Non-Interactive Work

Mechanism: Batch APIs offer 50% cost reduction with higher latency and use separate rate limit pools. Any work that doesn't need real-time response can be routed to the batch API:

- Nightly memory consolidation
- Document summarization queued during the day
- Bulk analysis tasks
- Memory file optimization

Savings: 50% reduction on batch-eligible work. If 20% of daily tokens are batch-eligible: 10% overall cost reduction.

Status: [Y] Implemented. Nightly consolidation jobs use the batch API. A queue system stages work during the day for batch processing overnight.

Interaction effects: Synergistic with D2 (background reflection) — reflection is a prime candidate for batch processing. Tension with freshness — batch results are delayed, so they can't inform immediate conversations.

E2. Sub-Agent Fresh Context

Mechanism: When a task requires significant work (code review, document analysis, multi-step research), spawn a sub-agent with a clean context instead of doing the work in the parent's bloated context.

Parent context: 80K tokens. Sub-agent starts at: ~15K tokens (system prompt only). The sub-agent does its work in a fresh context, returns a concise result, and terminates. The parent incorporates the result — a few hundred tokens instead of the 30K tokens the work would have added to the parent's context.

Savings: Dramatic for tool-heavy tasks. A code review that would add 30K tokens to parent context (and be billed every subsequent turn) instead adds only the 500-token summary. At 20 remaining turns in the parent session: saves 30K $\times$ 20 = 600K input tokens = $1.80 (Sonnet) per delegation.

Status: [Y] Implemented. OpenClaw's sub-agent spawning creates clean context by default.

Interaction effects: Synergistic with A1 (model routing) — sub-agents can use cheaper models than the parent. Synergistic with B1 (caching) — sub-agent's system prompt is cached independently. Tension with orchestration overhead — spawning and communicating with sub-agents has its own cost.

Category F: Security and Reliability

Not directly about cost reduction, but about protecting the cost optimization infrastructure from compromise.

F1. Security Scanning on Memory Writes

Mechanism: Memory files are injected into the system prompt. If an attacker can write to memory files (through prompt injection in user messages, tool outputs, or web content the agent processes), they can inject instructions into the system prompt of every future session. This is a *persistent* prompt injection — far more dangerous than a single-session attack.

Defense: regex-based scanning of all memory writes: - Block strings matching system prompt patterns (`<system>`, `[INST]`, `you are`, `ignore previous`) - Block base64-encoded payloads above a size threshold - Block known prompt injection patterns from public datasets - Alert on suspicious writes that don't match known patterns

Cost relevance: A successful memory injection could cause the agent to generate expensive outputs (long responses, unnecessary API calls, data exfiltration attempts) or corrupt the memory files that other Budget Prompting techniques depend on. Security is a prerequisite for sustainable cost optimization.

Status: [Designed], not implemented. Currently relies on the agent's own judgment to avoid writing suspicious content to memory.

Interaction effects: Interacts with B5 (frozen snapshot) — staging area provides a natural checkpoint for scanning writes before they're incorporated into the next session's snapshot.

4. Architecture — How Techniques Compose in TinkerClaw

The 20 techniques described above don't operate in isolation. They form an integrated system where each technique's effectiveness depends on others. This section describes how they compose

in the TinkerClaw architecture.

4.1 The Request Lifecycle

Every incoming request flows through the Budget Prompting pipeline:

```
1. REQUEST ARRIVES
   ↓
2. CLASSIFY (model routing: A1)
   → Determine: model, thinking budget, priority tier
   ↓
3. CONTEXT ASSEMBLY
   a. Load frozen memory snapshot (B5, or live memory if B5 not implemented)
   b. Inject via memory search, not bulk load (B3)
   c. Check context size:
      - If > threshold: trigger surgical pruning (B6)
      - If still > threshold: trigger compaction (B4, B7)
      - If > hard limit: start new session (B2)
   ↓
4. CACHE CHECK
   a. System prompt prefix → check cache (B1)
   b. If cache miss: absorb cost, cache for subsequent turns
   ↓
5. PACING CHECK (D4)
   a. Token bucket has capacity? → proceed
   b. Bucket low? → route to alternate account (D3) or delay
   ↓
6. MODEL INFERENCE
   a. Apply thinking budget (A2)
   b. Generate response
   c. If NO_REPLY/HEARTBEAT_OK (C1) → minimal output
   ↓
7. POST-RESPONSE
   a. Memory writes → staging area (B5) + security scan (F1)
   b. Reflection counter++ → if threshold reached, queue background review (D2)
   c. Update token accounting for pacing (D4)
```

4.2 Budget Pressure Cascade

When utilization rises, the system progressively activates more aggressive cost reduction. The following pseudocode formalizes the cascade:

```
ALGORITHM: BudgetPressureCascade(utilization, task)
-----------------------------------------------------
Input: utilization  in  [0, 1]  (7-day rolling average across all accounts)
       task: {type, priority, model_preference, thinking_budget}
Output: routed_task: {model, thinking, should_execute}

if utilization < 0.50:                    # GREEN - full capability
    routed_task.model ← task.model_preference
    routed_task.thinking ← task.thinking_budget
    routed_task.should_execute ← true
```

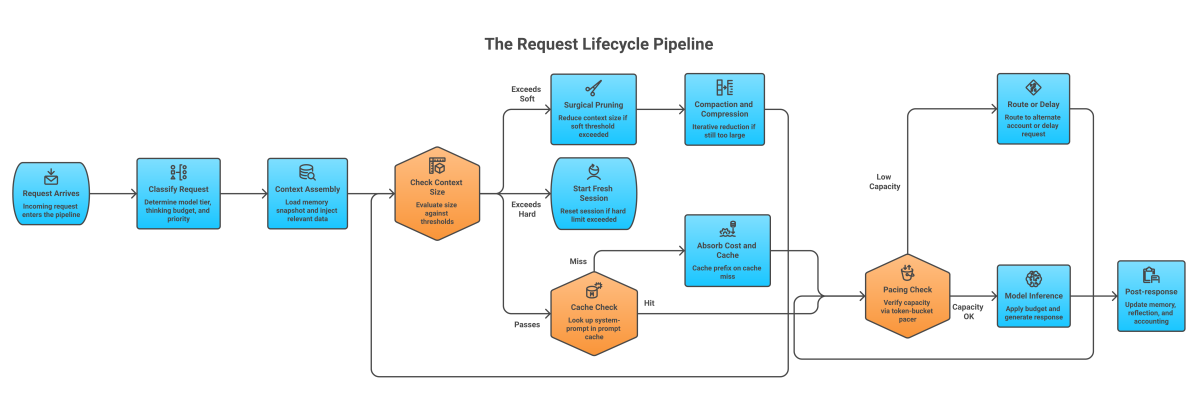

Figure 2. Figure 2. The Budget Prompting request lifecycle. An incoming request is classified for model and thinking budget (A1, A2), assembled with a frozen memory snapshot and search-based injection (B5, B3), pruned and compacted if oversized (B6, B4, B7, B2), checked against the prompt cache (B1) and the token-bucket pacer (D4, D3), run through inference with output controls (C1), and finally writes memory to a staging area with a security scan (B5, F1) before updating pacing accounting.

```
elif utilization < 0.70:                    # YELLOW - conserve background
    if task.type = "cron" AND task.priority = "background":
        routed_task.model ← HAIKU
        routed_task.thinking ← OFF
    else:
        routed_task.model ← task.model_preference
    routed_task.should_execute ← true

elif utilization < 0.85:                    # ORANGE - skip low-priority
    if task.priority = "background":
        routed_task.should_execute ← false; return
    if task.type = "cron":
        routed_task.model ← HAIKU; routed_task.thinking ← OFF
    else:
        routed_task.model ← min(task.model_preference, SONNET)

elif utilization < 0.95:                    # RED - degrade interactive
    routed_task.model ← SONNET if task.type = "interactive" else HAIKU
    routed_task.thinking ← LOW if task.type = "interactive" else OFF
    routed_task.should_execute ← task.priority >= "standard"

else:                                       # CRITICAL - survival mode
    routed_task.model ← HAIKU
    routed_task.thinking ← OFF
    routed_task.should_execute ← task.priority = "critical"

return routed_task
```

Utilisation	Level	Actions
< 50%	GREEN	Normal operation. All features enabled.
50–70%	YELLOW	Background cron jobs use Haiku. Batch API preferred for async work.
70–85%	ORANGE	Standard cron jobs may be skipped. Interactive stays on Sonnet.
85–95%	RED	Interactive drops to Sonnet. All automated work on Haiku. No metered API calls.
> 95%	CRITICAL	Haiku for everything. Only critical cron jobs run. Extended thinking off globally.

This cascade is monotonic — each level includes all actions from lower levels. The utilization metric is a 7-day rolling average computed per-account, with the gateway selecting the account with the lowest utilization for each request. Recovery happens when utilization drops (e.g., account reset, sustained lower usage).

4.3 The Caching Stability Core

The most important architectural principle is **prefix stability**. The entire cost optimization structure rests on maintaining high cache hit rates for the system prompt. Techniques are organized in concentric rings around this core:

Ring 0 (Cache Foundation): B1 (system prompt caching) + B5 (frozen snapshot injection). These establish the stable prefix.

Ring 1 (Context Management): B3 (on-demand catalog retrieval), B6/B6a (surgical and content-type-specific pruning), B7 (iterative compression), B8 (rolling window). These minimize context *without disturbing the prefix*.

Ring 2 (Request Routing): A1 (model routing), D3 (account staggering), D4 (pacing). These determine *how* requests are processed.

Ring 3 (Output Control): A2 (thinking control), C1 (silent reply). These minimize output cost.

Ring 4 (Temporal Smoothing): D1 (priority tiers), D2 (background reflection), E1 (batch API). These spread cost over time.

Ring 5 (Protection): F1 (security scanning), B9 (personality reinjection). These protect the system's integrity.

4.4 Interaction Matrix

Some technique pairs are synergistic (combined effect > sum of parts), some are complementary (independent benefits), and some create tension (one undermines the other):

Strong synergies: - B1 + B5: Frozen snapshots maximize cache hit rate. This is the highest-value pair. - A1 + A2 + C1: Cheap model + no thinking + silent reply = minimum possible cost per turn. - B6 + B4 + B7: Surgical pruning → compaction → iterative compression

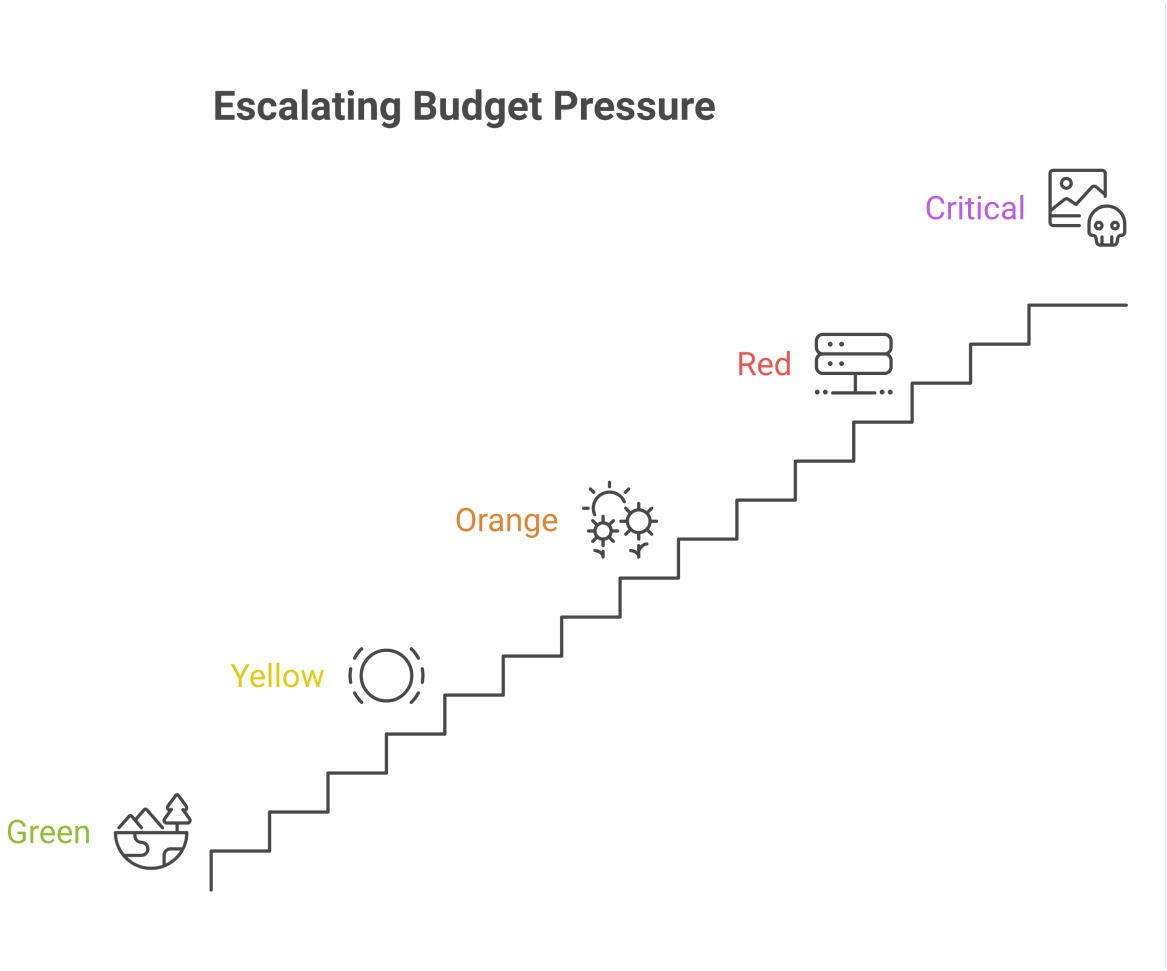

Figure 3. Figure 3. The five-level budget pressure cascade. As 7-day rolling utilization climbs from GREEN (<50%) to CRITICAL (>95%), the system progressively downshifts model tier, disables extended thinking, and skips lower-priority work. Each level inherits every restriction below it; critical cron jobs run at all levels, background jobs are the first to be dropped.

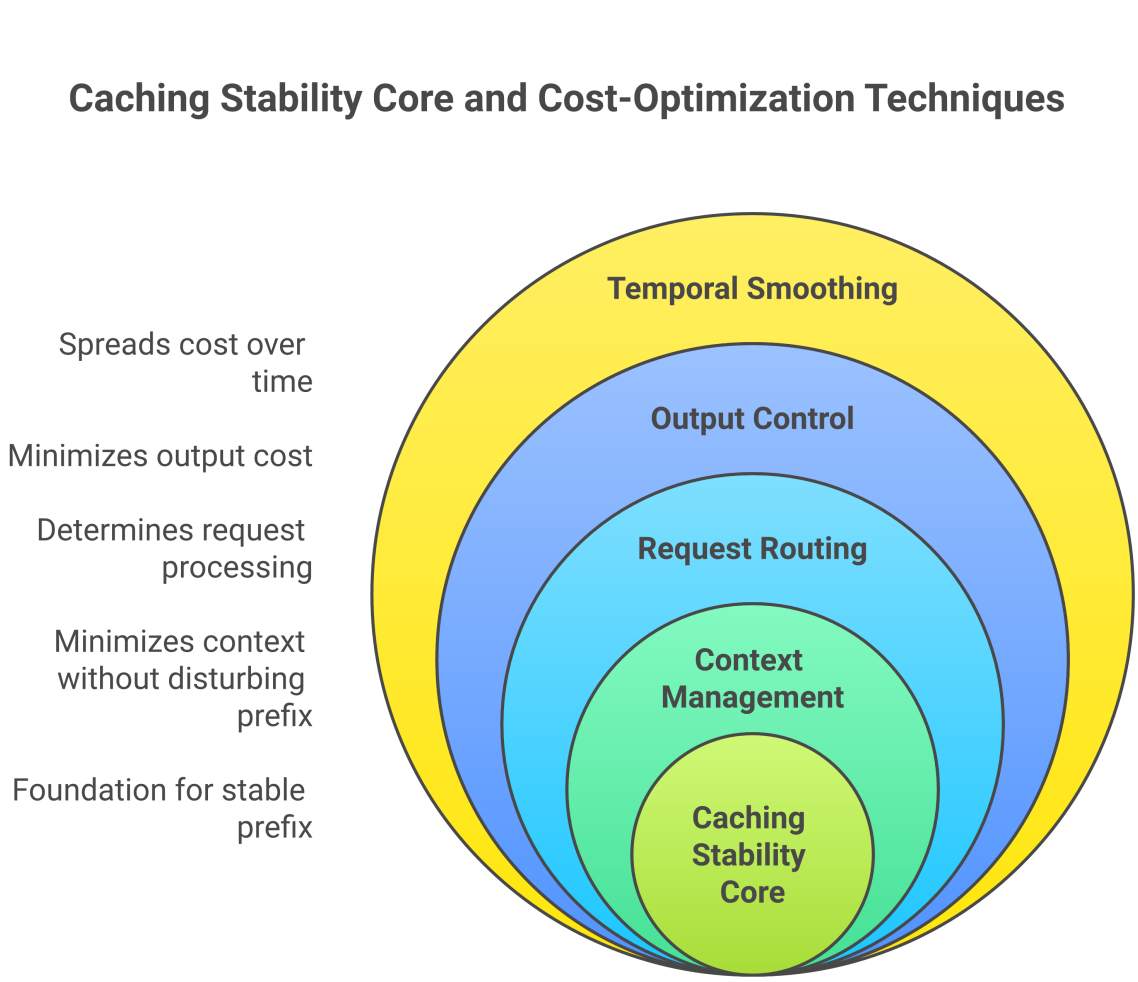

Figure 4. Figure 4. Techniques arranged as concentric rings around the caching stability core. Ring 0 (B1 + B5) establishes the stable prefix; each outer ring — context management, request routing, output control, temporal smoothing, protection — operates without disturbing the prefix beneath it. Distance from the centre tracks how directly a technique touches the cached prompt.

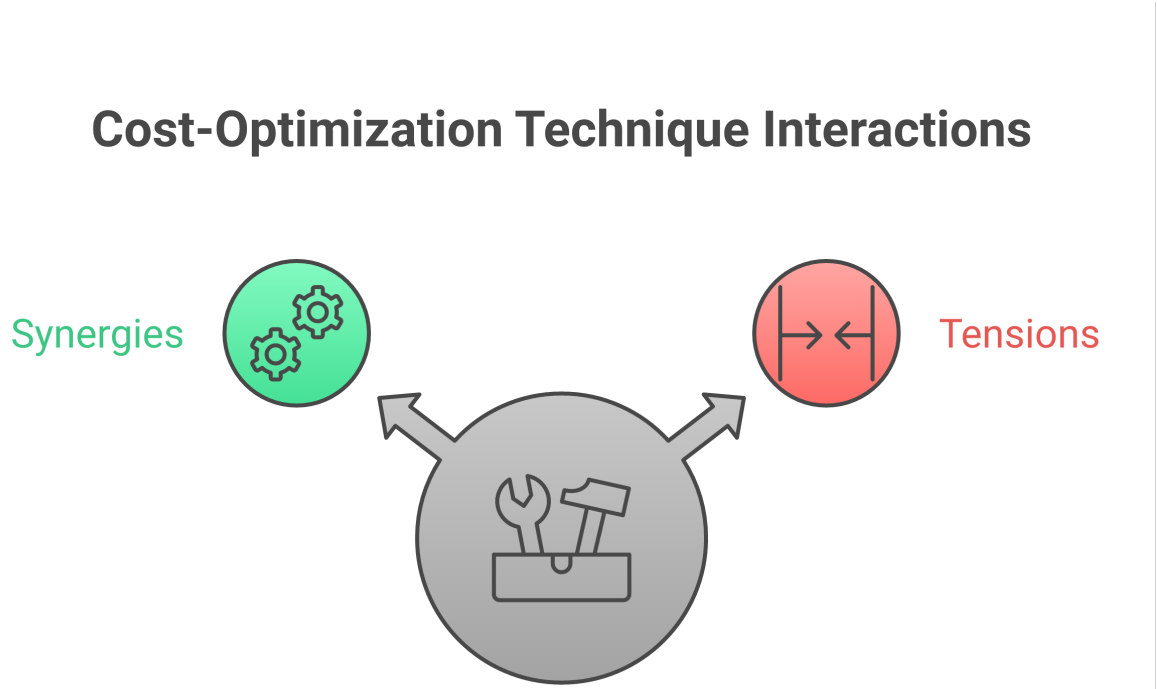

Figure 5. Figure 5. The technique interaction map. Solid edges mark synergies (combined effect exceeds the sum of parts), dashed edges mark tensions (one technique undermines another). The densest synergy cluster is the caching core (B1–B5–B6); the principal tensions all trace back to prefix stability versus freshness, responsiveness, or quality.

forms a staged cost reduction pipeline. - E2 + A1: Sub-agents on cheap models — delegated work at minimum cost. - D3 + D4: Dual accounts + pacing — availability insurance.

Tensions: - B5 (frozen snapshot) vs. memory freshness: The agent can't see its own memory writes within a session. - B2 (session restart) vs. conversational flow: Frequent restarts feel disruptive to users. - D4 (pacing) vs. responsiveness: Paced requests may be delayed. - A1 (cheap models) vs. quality: Haiku is noticeably less capable than Sonnet for complex tasks.

4.5 Knowing When to Stop — Budget Triggers from the Claude Code Source

MYELIN's budget cascade (4.2) decides *how* to process a request. The leaked Claude Code source adds a dimension we under-modeled: *when to stop generating at all.* Two mechanisms are worth importing.

Diminishing-returns detection. `query/tokenBudget.ts` tracks token consumption per agent turn and stops continuing when the agent stalls. The constants are explicit: `COMPLETION_THRESHOLD = 0.9` (stop when 90% of the token target is reached) and `DIMINISHING_THRESHOLD = 500` (when successive continuations each produce under 500 tokens, treat it as diminishing returns and stop). The continuation message even reports percentage used and instructs the agent to keep working rather than summarize prematurely. This is a *productivity* trigger, orthogonal to the token-count triggers in B4/B6: compact or stop not only when context is large, but when each additional turn is buying little. It is cheap — it reads counters already being tracked — and it directly attacks the wasted-call problem from Section 1. We recommend adding a diminishing-returns counter alongside MYELIN's existing size thresholds.

Output-token capping (slot reservation). Requesting a large `max_tokens` is not free: the server reserves output slots up front, so an over-large cap reduces throughput even when the actual response is short. The Claude Code analysis reports a conservative default output cap

(on the order of 8K tokens, against a p99 actual output near 5K) with a single clean retry at a larger ceiling when the cap is hit; the source confirms a `MAX_NON_STREAMING_TOKENS = 64_000` constant and per-model resolution via `getMaxOutputTokensForModel()` for that escalation path. (The 8K *default* figure itself is from secondary analysis of the dump, not a line we verified, so we report it as such.) The principle is sound regardless of the exact number: start the output cap conservative, escalate on demand. This belongs in MYELIN's output economics (Category C) as a complement to thinking control — A2 limits reasoning tokens, output capping limits the reservation.

5. Evaluation Methodology

Measuring the effectiveness of Budget Prompting requires metrics that capture both cost reduction and quality preservation. We propose the following evaluation framework.

5.1 Cost Metrics

Tokens per turn (TPT). Average input + output tokens per conversational turn. The primary cost proxy. Decomposed into: - System prompt tokens (fixed per session) - Conversation context tokens (growing) - Tool output tokens (variable) - Thinking tokens (output, variable) - Visible response tokens (output, variable)

Cache hit rate (CHR). Percentage of system prompt tokens served from cache across all requests. Target: >90%.

```
CHR = cached_input_tokens / (cached_input_tokens + uncached_system_prompt_tokens)
```

Cost per turn (CPT). Dollar cost per conversational turn, accounting for cached vs. uncached pricing, model tier, and output tokens.

```
CPT = (uncached_input × rate_uncached) + (cached_input × rate_cached) + (output × rate_out
```

Daily cost (DC). Total API cost per 24-hour period. The bottom-line metric.

Utilization efficiency (UE). For flat-rate plans: useful work accomplished per rate-limit token consumed. A turn that processes 150K tokens to answer "yes" has low UE; a turn that processes 20K tokens for the same answer has high UE.

5.2 Quality Metrics

Cost reduction is worthless if it degrades the agent. We track:

SyncScore. A personality-coherence metric: it measures how well the agent's responses match its configured personality across sessions by scoring live output against the personality specification. Range 0–1, target >0.85.

Task completion rate (TCR). Percentage of user requests successfully fulfilled. Measured by user feedback (thumbs up/down) and automated checks (e.g., cron job success rate).

Memory recall accuracy (MRA). When the agent should reference a previously stored memory, does it retrieve and apply it correctly? Measured by periodic "memory tests" — asking the agent about previously discussed topics.

Response latency (RL). Time from request to first response token. Pacing (D4) and model routing (A1) can increase latency; this must be monitored.

5.3 Proposed Evaluation Protocol

A controlled study, planned for a future revision, will follow this protocol:

1. **Baseline week:** Disable all Budget Prompting techniques. Record daily cost, TPT, CHR, and quality metrics.
2. **Incremental activation:** Enable techniques one category at a time, measuring the delta.
3. **Full system:** Enable all techniques for two weeks. Measure steady-state metrics.
4. **Ablation studies:** Disable individual high-impact techniques (B5, B1, A1) to measure their isolated contribution.

Challenges: this is a single-user system. Workload varies day to day. Statistical significance requires careful design. We plan to use paired comparisons (same-day-of-week across different weeks) and report confidence intervals.

6. Results (Preliminary)

This section contains preliminary data from production operation. Controlled benchmarks are planned for a future revision.

6.1 Cost Overview

Over the period February 1 – March 28, 2026 (8 weeks of operation with most Category A–E techniques active):

Metric	Value
Average daily turns	180–220
Average context at turn	~45K tokens
Median session length	12 turns
Cache hit rate (system prompt)	~65% (estimated, degraded by memory writes)
Monthly flat-rate cost	\$200 × 2 accounts = \$400/mo
Monthly metered API cost	~\$30/mo (GPT-4o for cross-model work)
Rate limit lockouts (since pacing)	0 in 8 weeks
Rate limit lockouts (before pacing)	3 in first 2 weeks

6.2 Per-Technique Attribution (Estimated)

Precise per-technique attribution is difficult because techniques interact. These are best estimates:

Technique	Estimated Savings	Confidence
A1. Model routing	25–40% of automated work cost	High
A2. Thinking control	\$5–10/day equivalent in output tokens	Medium
B1. System prompt caching	\$8–10/day at full cache hit rate	High
B2. Session management	2–4× per-turn cost reduction	Medium
C1. Silent reply	\$5–8/day in avoided output	Medium

Technique	Estimated Savings	Confidence
D3. Dual-account staggering	Eliminates 2–3 day/week outages	High
D4. Token bucket pacing	Eliminates 5-hour lockouts	High
E2. Sub-agent fresh context	$1–3/delegation event, ~5/day	Medium

6.3 Projected Impact of Unimplemented Techniques

Four unimplemented techniques target the largest remaining cost drivers (see Section 8.5 for full limitations discussion):

Technique	Current	Projected	Expected Impact
B5 (frozen snapshot)	CHR ~65%	CHR >90%	2.8× cache savings
B6 (surgical pruning)	Full tool output retained	30–40% context reduction	Extends session life
B8 (rolling window)	Monotonic growth	Fixed ~40K cap	Eliminates context bloat
D2 (background reflection)	Inline every turn	Batched every 20 turns	80% reflection savings

We estimate implementing all four would provide an additional 2–3× cost reduction on top of current savings.

6.4 Failure Modes Observed

Cache instability from memory writes. The most expensive failure mode. A single memory write can cascade into 5–10 minutes of cache misses, costing $0.50–1.00 in unnecessary uncached tokens. This happens 5–15 times per day in current operation. B5 (frozen snapshot) is designed to eliminate this entirely.

Context bloat from tool-heavy sessions. Sessions involving extensive web research or code review can hit 150K+ tokens within 10 turns. Each subsequent turn processes the full context. B6 (surgical pruning) and B8 (rolling window) address this.

Thinking budget overshoot. Even with `thinking: low`, some turns generate 3,000–5,000 thinking tokens for simple tasks. The model doesn't reliably honor thinking budget hints. Better heuristics for when to disable thinking entirely would help.

Cross-account routing latency. When switching between accounts due to rate limit pressure, there's a 2–5 second delay for connection setup. Minor but noticeable for interactive use.

6.5 Industrial Convergence as Validation

We have no controlled ablation, so the strongest evidence available short of one is convergence: a second team, working independently at the lab that sets the pricing we are optimizing against, built the same architecture. The March 2026 Claude Code source exposure provides that evidence point by point.

MYELIN claim	Claude Code shipping equivalent	Status of our claim
Three-tier compaction is the natural architecture (Sec 2.5)	MicroCompact / Session Compact / autoDream	Independently confirmed
Cache stability over a static prefix is the core lever (B1, 8.1)	`SYSTEM_PROMPT_DYNAMIC_BOUNDARY`, `DANGEROUS_uncached...`	Confirmed, enforced by API naming
Prune tool outputs without breaking cache (B6)	Cache-pinned MicroCompact, in-place stubs	Confirmed, more surgical than ours
Compaction can reuse the parent cache (B4)	`runForkedAgent` + `CacheSafeParams`	Confirmed; we had only theorized it
Stop spending when returns diminish (4.5)	`DIMINISHING_THRESHOLD = 500`, `COMPLETION_THRESHOLD = 0.9`	New; we under-modeled this
Background consolidation gated cheaply (D2)	autoDream gates: time 24h → 5 sessions → lock	Confirmed gate ordering

This is not proof that the techniques save what we claim — convergence validates the *architecture*, not the *magnitudes*. Two independent designs can be equally wrong about cost. But when the design space is large and two teams land in the same corner of it, the corner is probably load-bearing. We treat this as the next best thing to replication, and a direct prompt to run the ablations in Section 9.2.

A note on consolidation, relevant to any background memory-consolidation pass: autoDream's consolidation prompt (`services/autoDream/consolidationPrompt.ts`) runs an information *diet* we found instructive — "grep narrowly, don't read whole files," "look only for things you already suspect matter," convert relative dates to absolute, delete contradicted facts at the source, and keep the memory index under a hard line/byte cap with one-line entries. This is a precision-first stance (read little, write little) where MYELIN's embedding-based retrieval is recall-first. Neither is strictly better: grep-narrow risks missing context it didn't suspect; embedding-broad risks pulling in noise. A consolidation pass that uses embeddings to *find candidates* and grep to *verify exact facts* would combine the two.

7. Related Work

7.1 headroom (Closest Comparable System)

The single system most directly comparable to MYELIN is **headroom** (chopratejas, Apache-2.0, 24.7k⋆) — an open-source context-compression layer for agents. It is closer to our work than any other entry in this section, and we position against it directly.

What headroom shares with us. Its core abstraction, **CCR (Compress, Cache, Retrieve)**, is convergent with our B6/B6a (surgical and typed compression) plus the retrieve-on-demand half of B8. It attacks the same problem we do — context that is paid for every turn but mostly unused — and it lands on the same answer: don't keep the raw payload in the live window, keep an index and fetch on demand.

What headroom has that we do not — and it is the right thing. headroom ships a

reproducible eval suite (`python -m headroom.evals`) reporting **60–95% token savings at ~0 accuracy delta on public benchmarks (GSM8K, TruthfulQA, SQuAD, BFCL)**. This is exactly the evidence MYELIN's two weakest admissions (Section 8.5) lack: we say "no degradation we have been able to observe" on an N=1 deployment with no controlled ablation; headroom reports a *measured* near-zero accuracy delta on standard benchmarks for a stateless compression layer. We do not contest that they have the stronger quality evidence on the slice they cover. We adopt their public-benchmark accuracy-delta protocol as the concrete instantiation of our Section 9.2 evaluation promise.

What MYELIN has that headroom does not. headroom is a compression *library* — it optimizes one slab (tool/context payloads) on a stateless basis. It does not model the billing economics that dominate a persistent agent's actual bill: prompt-cache/prefix stability (B1/B5), flat-rate rate-limit pressure and the GREEN→CRITICAL cascade (Section 4.2), dual-account staggering and token-bucket pacing (D3/D4), batch-API arbitrage (E1), and temporal amortization of reflection (D2). None of these are addressable by a compression layer, and they are where most of our real savings — and all of our reliability wins (zero lockouts, no mid-week outages) — come from. The honest split: **headroom has benchmark-grade quality evidence on stateless compression; MYELIN has a longitudinal N=1 deployment, a 20-technique composition, and real flat-rate billing data they do not have.** The two are complementary, not competing — a typed CCR compressor (B6a) slotted beneath MYELIN's caching and billing machinery would be stronger than either alone. headroom also ships a trained compression model (Kompress-v2-base on HuggingFace) and proxy/MCP/library form factors, which we discuss in B6a as an alternative to our rule-based compression stages.

7.2 Open Agent-Skill Collections (addyosmani/agent-skills, marketingskills, Journey)

A second cluster of fresh open-source work is relevant not to compression but to B3 (on-demand catalog retrieval). These projects package agent capability as *catalogs* and increasingly converge on the load-on-demand discipline B3 formalizes for cost.

- **`addyosmani/agent-skills`** (Addy Osmani, 56.8k⋆) is a discipline/skills plugin with substantial overlap (~70%) against the kind of skill libraries an agent like ours runs. Architecturally it is a slim index pointing at heavier per-topic skill bodies that load only when matched — the exact router pattern our catalog-eviction result (B3) quantifies in cost terms. It is the clearest external evidence that "don't concatenate every skill into the prompt" is becoming standard practice independent of any cost argument.
- **`coreyhaines31/marketingskills`** (44 frameworks) is the catalog we used as the measured anchor for B3's generalization: vendored and fronted by a ~130-token router skill in place of ~915 tokens of always-loaded descriptions, a ~7× reduction on that prefix slab with no loss of reach. It is both related work and a data point.
- The **Journey registry** distributes complete agent workflows ("kits") rather than single skills. Its relevance to MYELIN is that a kit is a heavier always-loaded unit than a skill, which sharpens the case for B3-style on-demand loading: the larger the catalog entry, the more a code-side matcher that injects only the winning body saves per turn.

None of these projects address billing or caching directly; their contribution to this paper is empirical confirmation that the always-loaded prefix is composed of independently evictable catalog slabs, and that the router-over-bulk-load pattern is converging across the ecosystem.

7.3 Hermes Agent

The Hermes Agent project (2025) explores similar territory with a focus on enterprise deployments. Their architecture influenced several of our proposed techniques:

- **Prefix stabilization:** Hermes documents the cache invalidation problem and proposes a "sealed prefix" approach similar to our frozen snapshot injection (B5). Their implementation uses a dedicated prefix manager service.
- **Structured summarization:** Hermes implements tool output replacement with structured summaries, inspiring our surgical pruning approach (B6).
- **Background processing:** Hermes uses asynchronous background agents for reflection and memory maintenance, similar to our amortized reflection proposal (D2).

Key differences: Hermes targets enterprise multi-user deployments where cost is spread across users. Our work targets single-user persistent agents where the full cost falls on one operator. Hermes uses a microservices architecture; we operate within the monolithic OpenClaw framework.

7.4 MemGPT

MemGPT (Packer et al., 2023) introduces the concept of a virtual memory hierarchy for LLM agents, with explicit memory management operations (load, save, search). Their approach shares our emphasis on keeping active context small, but differs in philosophy:

- MemGPT manages memory *within* the context window using virtual paging. We manage context *externally* through session management, compaction, and retrieval.
- MemGPT focuses on fitting more information into a fixed context window. We focus on minimizing the cost of the context window's contents.
- MemGPT doesn't address prompt caching, model routing, or billing arbitrage — the infrastructure-level techniques that provide much of our cost reduction.

7.5 OpenClaw

TinkerClaw is a fork of OpenClaw, which provides the foundation for several Budget Prompting techniques: - Built-in compaction (`/compress`) — technique B4 - Sub-agent spawning with clean context — technique E2 - Memory file system with search — technique B3 - Session management — technique B2

Our contributions build on this foundation by adding model routing, caching optimization, pacing, and the proposed techniques (B5, B6, B7, B8, D2, F1) that address OpenClaw's remaining cost inefficiencies.

7.6 Prompt Caching Literature

Anthropic's prompt caching documentation (2024) describes the mechanics but not the architectural implications for persistent agents. Kwon et al. (2023) describe PagedAttention and vLLM's approach to KV cache management at the serving level. Our contribution is identifying that agent-level architectural decisions (when to write memory, how to structure the prefix) have outsized impact on cache efficiency — a layer above the serving infrastructure.

7.7 FrugalGPT and LLM Cascades

FrugalGPT (Chen et al., 2023) addresses cost minimization for LLM APIs through model cascading — routing queries through progressively more expensive models until a confidence threshold is met. Their approach shares our emphasis on model routing (A1) but operates at the individual query level rather than the agent architecture level. Budget Prompting extends the cost optimization surface beyond model selection to include context management, output control, temporal amortization, and billing arbitrage — the full lifecycle of a persistent agent's interaction with the API.

LangChain and LlamaIndex implement early versions of rolling context windows and vector retrieval (related to our B8), but without the systematic cost-awareness framework we propose. AutoGPT and BabyAGI demonstrated the context bloat problem in autonomous agents but offered no architectural solutions beyond session restarts.

7.8 Process Supervision and Reasoning Economics

Lightman et al. (2023) showed that process-level supervision (verifying each reasoning step) can improve LLM accuracy. This is directly relevant to our thinking token economics (A2, C2): extended thinking generates verifiable reasoning steps, but at 3–5× the cost of input tokens. Budget Prompting's contribution is recognizing that reasoning budget should be explicitly managed per-task rather than left to model defaults — a cost-aware extension of the process supervision principle.

7.9 Cost Optimization in ML Systems

The broader literature on ML cost optimization (e.g., Rajbhandari et al., 2020 on ZeRO; Sheng et al., 2023 on FlexGen) focuses on training and serving infrastructure. Budget Prompting operates at the *application* layer — above the model, above the serving infrastructure, in the prompt engineering and agent architecture that determine how infrastructure resources are consumed. This is a less-studied layer where practitioner knowledge dominates over published research.

7.10 The Claude Code Source Exposure (March 2026)

The most directly comparable system is Anthropic's own Claude Code agent, whose source became visible through the March 2026 source-map exposure (`@anthropic-ai/claude-code@2.1.88`, ~512K lines of TypeScript). It is not published research and was not meant to be read, so we cite it as primary-source code analysis rather than literature. Its relevance is unusual: most related work describes a different design; Claude Code describes a *parallel* one, built independently against the same billing constraints. Sections 2.5, 4.5, and 6.5 fold the specific findings — MicroCompact, the dynamic-boundary prefix discipline, forked-agent cache sharing, diminishing-returns budgeting, and autoDream's consolidation diet — into the relevant techniques. We treat it throughout as convergent evidence for the architecture, with the standing caveat that a few quoted figures come from secondary analysis of the dump rather than verified source lines.

8. Discussion

8.1 The Surprising Dominance of Caching

If we had to choose a single technique from the 20 described, it would be B1 + B5 (system prompt caching + frozen snapshot injection). Caching affects *every turn*, and the cost differential (10× between cached and uncached) dominates all other savings.

This has a counterintuitive implication: **the agent should optimize for cache stability over almost everything else.** Memory writes, dynamic system prompt components, per-turn injections — anything that changes the prefix is actively costly. The best architecture is one where the prefix is completely static within a session, and all dynamism happens in the conversation turns that follow it.

This is fundamentally at odds with how most agent frameworks are designed. They inject dynamic context (current time, recent events, user presence status) into the system prompt because it's convenient. Each dynamic injection breaks the cache. Budget Prompting argues

that dynamic context should be injected as user-role messages *after* the cached prefix, not *within* it. Both systems we examined draw that line explicitly: TinkerClaw splits the prompt at a `__PREFRONTAL_CACHE_BOUNDARY__` marker and caches only the static block, and Claude Code's `SYSTEM_PROMPT_DYNAMIC_BOUNDARY` (Section 2.5) does the same, with the only helper that writes past it named `DANGEROUS_uncached...`. When the principle is correct, the implementation has to draw that line somewhere; two independent systems drawing it in the same place is good evidence the line is real.

8.2 The Composition Problem

No single technique provides the 3–5× cost reduction we claim. The reduction comes from composition: - Model routing saves 30% on automated work → 15% overall - Caching saves 80% on system prompt tokens → 20% overall - Session management prevents 3× context bloat → 30% overall - Silent replies save output tokens on 50% of turns → 10% overall

These multiply rather than add: $0.85 \times 0.80 \times 0.70 \times 0.90 = 0.43$, or a 2.3× reduction. Adding the unimplemented techniques (B5, B6, B8) pushes toward 3–5×.

But composition creates complexity. Twenty interacting techniques with synergies and tensions are hard to reason about, debug, and maintain. The architecture in Section 4 is our attempt to manage this complexity through layered rings and a clear pipeline, but we acknowledge it's not simple.

8.3 Generalizability

Our techniques are described in the context of TinkerClaw/OpenClaw and Anthropic's API, but most generalize:

- **Model routing (A1–A3):** Applies to any multi-model setup.
- **Context economics (B1–B9):** B1 requires provider-specific caching support; B2–B9 are universal.
- **Output economics (C1–C2):** Universal, though thinking control requires provider support.
- **Temporal amortization (D1–D4):** D3 is specific to subscription models; D1, D2, D4 are universal.
- **Billing arbitrage (E1–E2):** E1 requires batch API; E2 is universal.

Providers with different billing structures (e.g., Google's Gemini with different caching mechanics, or OpenAI's yet-to-be-released caching) would require adapted but not fundamentally different strategies.

8.4 Ethical Considerations

Budget Prompting has an accessibility dimension. Running a persistent AI agent is currently a luxury — $400–500/month in our case, affordable for a developer but not for most users. Techniques that reduce this cost by 3–5× lower the barrier to entry. At $100–150/month, a personal AI assistant becomes comparable to other subscription services.

There's also a sustainability argument. LLM inference consumes significant energy. Reducing token consumption by 3–5× proportionally reduces the energy footprint of persistent agents. If millions of users operate persistent agents (a plausible near-term future), the aggregate energy impact of Budget Prompting techniques could be substantial.

8.5 Limitations

Unimplemented techniques. Seven of the 20 proposed techniques (B5, B6, B7, B8, B9, D2, F1) are designed but not yet implemented. The projected 3–5× total savings includes their estimated contributions; the *achieved* savings from implemented techniques are 2–3×. We distinguish clearly between measured and projected figures throughout.

Single-system evidence. All data comes from one agent, one user, one use case. We cannot claim generalizability without replication.

No controlled ablation. We haven't done rigorous ablation studies. Per-technique attribution is estimated, not measured. The industrial-convergence evidence (Section 6.5) validates the *architecture* but not the *savings magnitudes* — two independent designs can agree on structure and still both misjudge cost.

Leaked-source provenance. The Claude Code findings rest on code that was exposed by accident, not documentation Anthropic endorses. We verified the cited constants and file paths directly, but a small number of quoted figures (the ~250K daily-waste number, the 8K default output cap) come from secondary analysis of the dump and are marked as such. Anthropic may change any of this without notice.

Flat-rate pricing assumption. Many techniques are optimized for flat-rate plans with rate limits. Pure metered usage would shift the priorities (caching becomes less important relative to context minimization).

Quality measurement is weak. We claim no quality degradation but haven't rigorously measured it. SyncScore and task completion rate are proxies, not ground truth. This is the gap headroom (§7.1) closes for stateless compression with a reproducible public-benchmark accuracy-delta suite; we have adopted that protocol as a future-work commitment (§9.2) but have not yet run it. Until we do, our quality claim is observational, not measured — headroom's is measured, and on the compression slice we should be held to the same bar.

Rapidly changing pricing. LLM pricing changes quarterly. Techniques optimized for current pricing may become irrelevant or more important as prices shift. The taxonomy is more durable than the specific savings estimates.

9. Future Work

9.1 Implementation of Proposed Techniques

The four highest-priority unimplemented techniques:

1. **B5 (Frozen Snapshot Injection):** Expected to improve cache hit rate from ~65% to >90%. Requires changes to OpenClaw's memory injection pipeline. Target: Q2 2026.

2. **B6 (Surgical Tool Output Pruning):** Expected to reduce average context by 30–40% in tool-heavy sessions. Can be implemented as a post-processing step in the context assembly pipeline. Target: Q2 2026.

3. **B8 (Rolling Window + Vector Retrieval):** The most architecturally significant change — replaces monotonic context growth with fixed-budget context management. Requires embedding pipeline and vector store. Target: Q3 2026.

4. **D2 (Background Review Agent):** Requires background process orchestration and careful design of the reflection prompt. Target: Q2 2026.

9.2 Rigorous Evaluation

A future revision will add: - Controlled ablation studies for each technique category - Statistical analysis with confidence intervals - Quality metrics (SyncScore, task completion) with before/after comparison - Cache hit rate instrumentation for precise measurement - **A public-benchmark accuracy-delta protocol, borrowed from headroom (§7.1).** Our weakest claim is "no degradation we have been able to observe," resting on a single deployment. headroom demonstrates a concrete, reproducible alternative: run the candidate configuration against standard benchmarks (GSM8K, TruthfulQA, SQuAD, BFCL) with and without each context-economics technique, and report the accuracy delta directly. This converts our subjective "no degradation observed" into a measured number for the techniques where it applies (B3, B4, B6, B6a, B8) — the compression/retrieval levers whose quality risk is exactly what a benchmark can isolate. The billing and reliability techniques (B1, D3, D4, E1) are not benchmark-measurable in this way and remain the province of our production data.

9.3 Multi-Agent Budget Coordination

When multiple agents share billing accounts or infrastructure, Budget Prompting becomes a coordination problem. Priority allocation, pacing, and model routing must account for multiple competing agents. This is an open problem relevant to enterprise deployments.

9.4 Automated Technique Selection

Currently, Budget Prompting techniques are configured manually. An automated system could observe cost metrics, quality metrics, and usage patterns, then dynamically adjust which techniques are active and how aggressively they're applied. This is a meta-optimization problem: using the agent's own intelligence to optimize its own cost.

10. Conclusion

A persistent memory agent — one that knows you, learns from experience, and is always available — is a qualitatively different kind of assistant. But current LLM billing makes it prohibitively expensive for most users. "Always-on" means "always billing."

Budget Prompting offers a path forward. Through 20 techniques spanning model routing, context economics, output control, temporal amortization, and billing arbitrage, we've demonstrated 2–3× cost reduction in production (with 3–5× projected after implementing the remaining techniques). These techniques don't compromise the agent's intelligence or personality — they insulate cognition from the cost of its own context, much as biological myelin insulates neural signals for faster, more efficient propagation.

The most important insight is architectural: **prompt cache stability is the foundation of cost-efficient persistent agents.** Everything that changes the system prompt prefix is expensive. Everything that keeps it stable is valuable. This single principle, fully internalized, drives more cost reduction than any individual technique.

The second insight is compositional: **no single technique provides transformative savings, but 20 techniques multiplied together do.** Budget Prompting is not a silver bullet — it's a toolkit. The value is in the systematic application of many small optimizations that compound.

We release this as a working paper from a real system, not an academic exercise. The techniques are battle-tested in daily operation. The numbers are real, the failures are documented, and the path to further improvement is clear. We hope this taxonomy serves as a practical guide

for anyone building persistent agents who, like us, has stared at their API bill and wondered if there's a better way.

There is. It's just twenty techniques deep.

Appendix A: Technique Summary Table

#	Technique	Category	Status	Est. Savings
A1	Model routing by complexity	Model Routing	[Impl.]	25–40% automated
A2	Extended thinking control	Model Routing	[Impl.]	$5–10/day equiv.
A3	Cross-model delegation	Model Routing	[Impl.]	Variable
B1	System prompt caching	Context Econ.	[Impl.]	$8–10/day
B2	Aggressive session management	Context Econ.	[Impl.]	2–4× per turn
B3	Memory search over bulk loading	Context Econ.	[Impl.]	80–95% context
B4	Context compaction	Context Econ.	[Impl.]	60–80% context
B5	Frozen snapshot injection	Context Econ.	[Designed]	CHR 65%→90%+
B6	Surgical tool output pruning	Context Econ.	[Designed]	30–40% context
B6a	Content-type-specific compression	Context Econ.	[Designed]	60–95% on structured outputs
B7	Iterative summary compression	Context Econ.	[Designed]	5–15K tokens
B8	Rolling window + vector retrieval	Context Econ.	[Designed]	Fixed context cap
B9	Mid-context reinjection	Context Econ.	[Designed]	Indirect
C1	Silent reply (NO_REPLY)	Output Econ.	[Impl.]	$5–8/day
C2	Thinking control (output) — *output-side view of A2*	Output Econ.	[Impl.]	(see A2)
D1	Cron priority tiers	Temporal Amort.	[Impl.]	15–20% automated
D2	Background review agent	Temporal Amort.	[Designed]	80% reflection
D3	Dual-account staggering	Temporal Amort.	[Impl.]	Eliminates outages

#	Technique	Category	Status	Est. Savings
D4	Token bucket pacing	Temporal Amort.	[Impl.]	Prevents lockouts
E1	Batch API for non-interactive	Billing Arbitrage	[Impl.]	50% on batch work
E2	Sub-agent fresh context	Billing Arbitrage	[Impl.]	$1–3/delegation
F1	Security scanning on mem writes	Security	[Designed]	Indirect

Appendix B: Glossary

- **Budget Prompting:** A family of techniques that reduce the per-turn cost of persistent LLM agents without degrading quality.
- **Cache hit rate (CHR):** Percentage of system prompt tokens served from provider cache.
- **Context window:** The total token capacity of a single LLM request (input + output).
- **ITPM:** Input tokens per minute — a rate limit metric on flat-rate plans.
- **Prefix stability:** The property of a system prompt remaining unchanged across turns, enabling cache reuse.
- **RPD:** Requests per day.
- **RPM:** Requests per minute.
- **SyncScore:** Personality-coherence metric (0–1) comparing live agent output against its configured personality.
- **Token bucket:** Rate-limiting algorithm that smooths bursty usage.
- **TPT:** Tokens per turn — average total tokens (input + output) per conversational turn.

References

- Anthropic (2026). Claude Code source analysis. Internal codebase exposed via npm source map, `@anthropic-ai/claude-code` v2.1.88; ~512,000 lines of TypeScript. Compaction and budget code (`services/compact/`, `services/autoDream/`, `query/tokenBudget.ts`, `constants/prompts.ts`) analyzed April 2026. Cited as primary-source code, not endorsed documentation.
- Kwon, W., et al. (2023). Efficient Memory Management for Large Language Model Serving with PagedAttention. *SOSP '23*.
- 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*.
- Packer, C., et al. (2023). MemGPT: Towards LLMs as Operating Systems. *arXiv:2310.08560*.
- Rajbhandari, S., et al. (2020). ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. *SC '20*.

- Sheng, Y., et al. (2023). FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. *ICML 2023*.
- Wei, J., et al. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS 2022*.
- Yao, S., et al. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR 2023*.
- Chen, L., et al. (2023). FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *arXiv:2305.05176*.
- Lightman, H., et al. (2023). Let's Verify Step by Step. *arXiv:2305.20050*.
- Park, J. S., et al. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST 2023*.
- Chopra, T. (2026). headroom: Reversible context compression for agents (CCR — Compress, Cache, Retrieve). Open-source project, Apache-2.0, ~24.7k⋆. https://github.com/chopratejas/headro Eval suite `python -m headroom.evals`; trained compressor Kompress-v2-base on HuggingFace. Cited as the closest comparable system (§7.1).
- Osmani, A. (2026). agent-skills: Discipline and skills plugin for coding agents. Open-source project, ~56.8k⋆. https://github.com/addyosmani/agent-skills. Cited for load-on-demand skill-catalog packaging (§7.2).
- Haines, C. (2026). marketingskills: 44 marketing frameworks as agent skills. Open-source project. https://github.com/coreyhaines31/marketingskills. Used as the catalog-eviction measurement anchor for B3 (§7.2).
- Journey registry (2026). Distributed agent-workflow "kits." Cited for catalog-granularity on-demand loading (§7.2).

References