NeuraGrowth

Context Budget Playbook: Cut Agent Token Costs by 60%

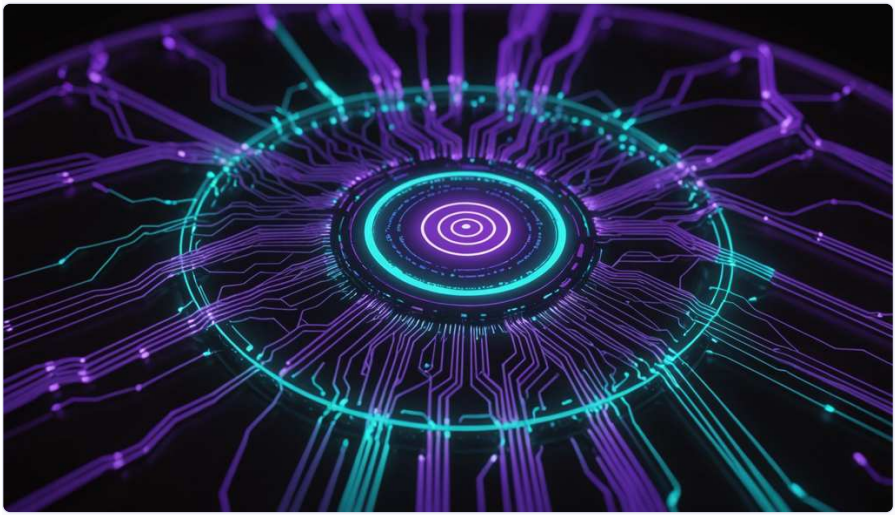
The engineering manager's field guide to auditing, compressing, and caching LLM context at production scale.

NEURAGROWTH.CO

2026-06-11

CONTENTS

01 Why Context Costs Are Eating Your AI Budget



The Invisible Line Item

Most engineering teams ship their first AI agent, watch the demos go well, then get their first AWS or GCP bill and feel a quiet panic. The compute cost for inference is not the surprise - the token volume is. Context windows have grown from 4K to 200K tokens, and that scaling is not free. Every token you send to the model is a billable event.

Anthropicss published pricing (as of early 2025) for Claude 3.5 Sonnet sits at $3.00 per million input tokens and $15.00 per million output tokens. For Claude 3 Haiku, those numbers drop to $0.25 and $1.25. The difference matters enormously at scale, but even Haiku costs real money when your agent is processing 50,000 tokens per task across 10,000 daily tasks.

The Math That Surprises People

Consider a customer support agent with these characteristics:

Parameter	Value
Average context per call	48,000 tokens
Daily call volume	8,000
Model	Claude 3.5 Sonnet
Input cost	$3.00 / 1M tokens

Daily input token spend: 48,000 x 8,000 = 384,000,000 tokens = **$1,152/day**
Annualized: **$420,480** - just for input tokens, before output, before retries, before evaluation runs.

Now apply a 40% context reduction. Same daily volume. Same model. Cost drops to $691/day, saving **$168,000/year** - from one lever.

Where the Fat Actually Lives

Context bloat is not random. In production agent systems, token waste clusters in predictable places:

1. **System prompt repetition** - The same 2,000-word system prompt prepended to every single turn, including turns where 80% of it is irrelevant.

2. **Tool result verbosity** - A search API returning 15,000 tokens of raw HTML when the agent needed three sentences.
3. **Conversation history accumulation** - Multi-turn agents naively appending every exchange, with turn 1 still visible in turn 47.
4. **Redundant retrieved chunks** - RAG pipelines retrieving five 500-token chunks that overlap semantically by 70%.
5. **Scratchpad pollution** - Chain-of-thought traces and intermediate reasoning stored in the context indefinitely.

The Economic Frame

This guide treats context optimization as a **capital allocation decision**, not a technical preference. Every optimization strategy in the following sections has a cost (engineering hours to implement) and a return (dollars saved per month). You will build a model that lets you rank those investments and defend them to finance or leadership.

The strategies in this playbook routinely deliver 40-60% reductions in context volume without measurable quality degradation. Some teams report up to 70% on document-heavy pipelines after combining techniques. The 60% headline is conservative and achievable within a single sprint for most production systems.

What This Guide Assumes

- You are running AI agents in production or late-stage staging against a paid API.
- You have access to API call logs or can instrument your agent framework to capture token counts per call.
- Your team has at least one engineer who can modify prompt templates and agent loop logic.

- You are comfortable reading Python-style pseudocode. Full working implementations are outside scope, but every technique here is concrete enough to implement the same week you read it.

WHY CONTEXT COSTS ARE EATING YOUR AI BUDGET

02

Context Cost Audit: Measure Before You Cut

You Cannot Optimize What You Cannot See

Before touching a single prompt, instrument your system to capture the exact distribution of token usage. Most teams skip this step, guess at the biggest problem, and optimize the wrong thing. A two-hour instrumentation pass will tell you where 80% of your token spend lives.

Step 1: Capture Per-Call Token Breakdowns

Every major inference provider returns token counts in the API response. Make sure you are logging them.

```
# Example: capturing token usage from Anthropic SDK response
response = client.messages.create(
 model="claude-3-5-sonnet-20241022",
 max_tokens=4096,
 messages=messages
)

token_log = {
 "call_id": call_id,
 "task_type": task_type,
 "input_tokens": response.usage.input_tokens,
 "output_tokens": response.usage.output_tokens,
 "cache_read_tokens": response.usage.cache_read_input_tokens,
# if cache enabled
 "cache_write_tokens":
response.usage.cache_creation_input_tokens,
 "timestamp": datetime.utcnow().isoformat()
}
logger.info(json.dumps(token_log))
```

Log every call. Do not sample. You need the full distribution, not just averages.

Step 2: Segment by Task Type

Aggregate your logs by task type or agent role. You are looking for this kind of table after 24-48 hours of production data:

Task Type	Avg Input Tokens	Avg Output Tokens	Call Volume/ Day	Daily Input Cost
Customer triage	12,400	380	4,200	$156
Document summarize	67,000	1,200	800	$161
Code review agent	28,000	2,400	1,100	$92
Retrieval Q&A	19,500	600	3,000	$175

This immediately shows you that document summarization and retrieval Q&A are your high-value targets, even though document summarization has lower call volume - the per-call cost is extreme.

Step 3: Decompose Context by Component

For your top two or three task types by cost, break down what is actually inside the context. Add debug logging that labels context segments:

```
def build_context_with_labels(system_prompt, history, retrieved_chunks, tool_results):
 segments = {
 "system_prompt": count_tokens(system_prompt),
 "conversation_history": count_tokens(history),
 "retrieved_chunks": count_tokens(retrieved_chunks),
 "tool_results": count_tokens(tool_results)
 }
 log_context_breakdown(segments)
 return assemble_messages(system_prompt, history, retrieved_chunks, tool_results)
```

A typical breakdown in a naive RAG agent might look like:

Segment	Tokens	% of Context
System prompt	2,100	11%
Conversation history	8,400	43%
Retrieved chunks	6,800	35%
Tool results	2,200	11%

This tells you that conversation history is your primary target - 43% of context, and likely the most compressible.

Step 4: Compute Tokens-Per-Task and Cost-Per-Task

For multi-turn agents, sum all turns in a single task session to get your real cost unit:

```
# tokens_per_task = sum of input tokens across all turns in one agent session
tokens_per_task = sum(call['input_tokens'] for call in session_calls)
cost_per_task = (tokens_per_task / 1_000_000) * INPUT_PRICE_PER_MILLION
```

Benchmark this against what the task is worth. A customer support resolution worth $4.20 in agent value with a $1.85 token cost is fine. A $0.12 value task with a $0.95 token cost is your immediate priority.

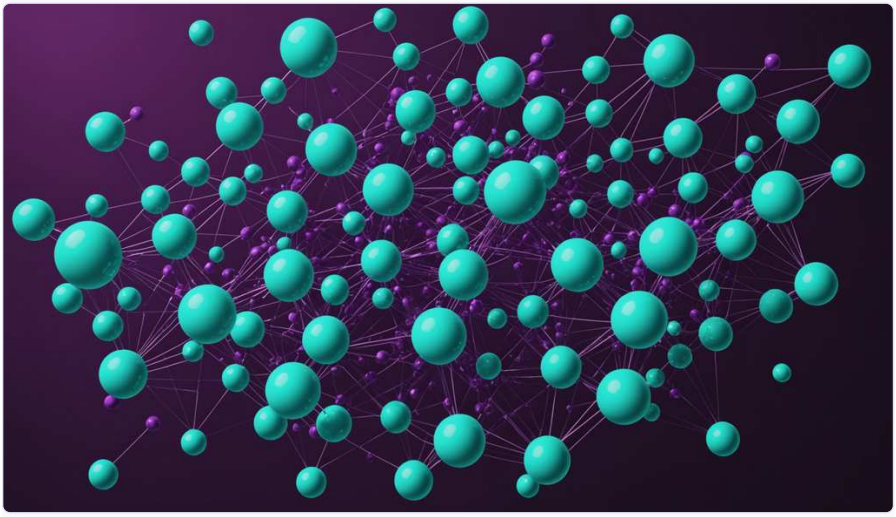
Step 5: Build a Hotspot Matrix

Score each task type on two axes: cost per task and daily call volume. Plot them on a 2x2. High cost per task, high volume = fix this week. High cost per task, low volume = fix next sprint. Low cost per task, high volume = monitor. Low cost per task, low volume = ignore.

This matrix becomes your roadmap for the optimization work in the chapters ahead. Bring it to your next sprint planning session. It is the artifact that turns "we should reduce token costs" from a vague goal into a prioritized backlog.

03

Semantic Deduplication: Remove Redundant Context at Retrieval Time



The Problem: 70% Overlap in Your RAG Chunks

In most retrieval-augmented generation pipelines, the top-k retrieved chunks overlap heavily. If you retrieve the top 5 chunks for a query about a product's refund policy, there is a high probability that chunks 2, 3, and 4 say the same thing in slightly different words - because your source documents repeat themselves, your chunking strategy cut across paragraph boundaries, or you indexed multiple versions of the same page.

You pay for every token in all five chunks. But the model gets roughly the same information it would from one and a half chunks.