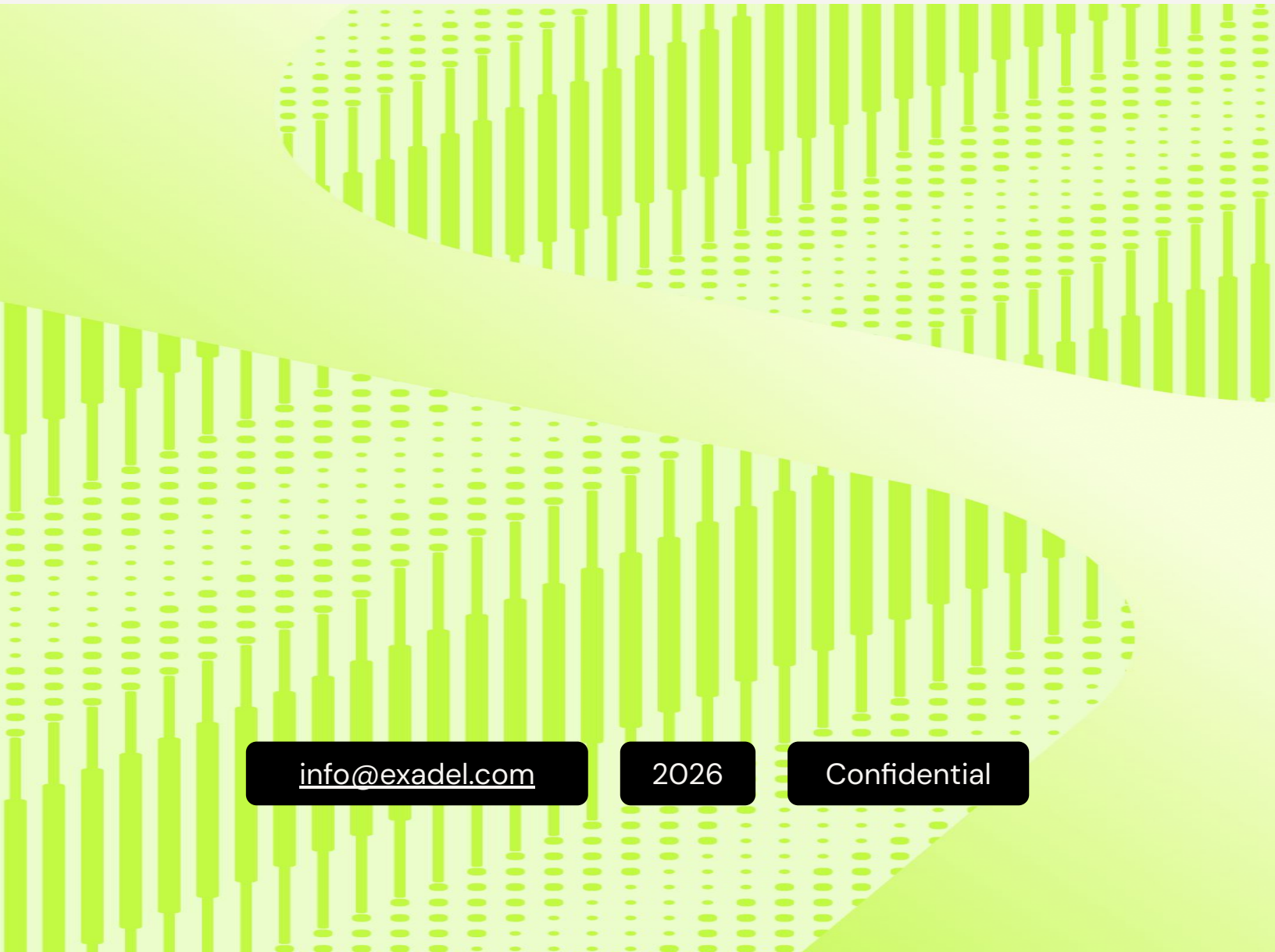
EXADEL AI

Token Cost Optimization

A Practical Framework for Enterprise AI Teams



info@exadel.com

2026

Executive Summary

AI tools, solutions, and agents are **essential companions** in many enterprise roles. Skyrocketing use of AI, while delivering cost and productivity gains, is also bringing another critical issue into focus: all AI usage comes at a price, and the bills are starting to **add up**.

Balancing the need to maintain and accelerate AI adoption with elevating costs is quickly becoming a new fiscal priority, as **AI inference bills** have a way of ticking up before enterprises can adjust. It's not difficult to see why AI sticker shock is becoming more common. An enterprise team, for example, spins up a promising agent, runs it through a few realistic workflows, and suddenly the monthly budget projection looks very different from the prototype. The root cause is almost always the same: tokens. More specifically, too many of them are being processed in too many redundant ways. Repeat these steps a few dozen times and the cumulative costs can be substantial.

Adopting token cost optimization is an effective method to address this challenge, but it is not a single technique. It is a discipline that spans how enterprise AI users structure prompts, the AI models invoked for specific tasks, how context is managed across multi-turn interactions, and how agent workflows are architected at the system level. Getting it right in our experience can reduce inference spend by 50 to 80 percent without sacrificing quality. Getting it wrong means paying full price for work a model has already done.

This white paper maps the landscape of available token optimization methods so engineering, product, and AI leaders can make informed decisions about where to start, and what elements can be incorporated into a full framework for better, more predictable AI spend.

Why Token Costs Compound Faster Than Expected

Before exploring solutions, it helps to understand why costs escalate so quickly in production.

LLM pricing is straightforward in isolation: input tokens and output tokens, billed per million. But in agentic systems, those numbers stack in ways that catch enterprise AI teams off guard. A long system prompt of 10,000 tokens, sent on every API call across a 50-turn agent session, contributes 500,000 input tokens before the model generates a single word of output. Multiply that across concurrent users and the math shifts fast.

The Anthropic Claude pricing page, for example, illustrates the range well: Claude Opus 4 series models run at 15 per million input tokens and 75 per million output tokens at standard rates, while Haiku 3 sits at 0.25 and 1.25 respectively. That is a 60x spread between the most capable and most economical tiers. Model selection alone is one of the highest-leverage levers available for token optimization.

Beyond model choice, there are four main categories of optimization worth mastering: prompt caching, prompt compression and design, model routing, and agentic architecture.

Prompt Caching: Paying Once for Repeated Context

Prompt caching is the most impactful optimization for applications that send the same content repeatedly, and in agentic systems, that is nearly universal. System prompts, retrieved documents, tool schemas, and conversation history all tend to remain stable across calls within a session while only a small slice of dynamic content changes.

The mechanics work at the KV (key-value) layer of the transformer. During the prefill phase, the model computes attention representations for every input token. Caching stores those representations so subsequent requests that share the same prefix can skip recomputation entirely. Cache reads cost roughly 10% of standard input pricing, meaning a single cache hit after a 5-minute write already breaks even.

Research from PwC published in January 2026 evaluated prompt caching across OpenAI, Anthropic, and Google over 500 agent sessions with 10,000-token system prompts. The results were compelling: **prompt caching reduced API costs by 41 to 80 percent and improved time-to-first-token by 13 to 31 percent** across all providers tested. Critically, the study found that naive full-context caching could paradoxically increase latency when dynamic tool results were included in the cached prefix. The winning strategy was targeted: cache the stable system prompt, and place dynamic content at the end of the prompt where it does not invalidate earlier cache blocks.

Practical guidance for teams implementing prompt caching:

Anchor stable content at the start

System instructions, static context, and reference documents should appear before any dynamic content. Cache invalidation flows from the first-changed token onward, so dynamic fields at the end preserve the most cache hits.

Separate tool results from cached prefixes

Including tool call outputs in a cached block creates a rapidly changing prefix that triggers constant rewrites. Excluding them from the cache boundary improves both cost and latency consistency.

Understand provider-specific minimums

Cache benefits only activate once a token threshold is crossed. For shorter prompts, the cost benefit may be marginal until context length grows.

For RAG-heavy applications, newer caching research offers an additional frontier. The **KV Packet framework from researchers at TU Munich and TU Darmstadt** proposes wrapping document blocks in lightweight trained adapters so precomputed KV states can be reused across different query contexts without recomputation. Standard KV caches are context-dependent, meaning a cached document's representations become invalid when the surrounding context changes. KV Packet addresses this by making cached documents effectively context-independent, enabling true plug-and-play cache reuse in RAG pipelines. This remains an active research area, but teams running high-volume document retrieval workflows should watch it closely.

Prompt Compression and Output Efficiency

Tokens spent on verbosity are tokens wasted. This applies to both inputs and outputs, and the optimization strategies for each differ.

Input compression targets the prompt itself. Long system prompts accumulate over time as edge cases are patched, instructions expand, and context grows. Techniques include:

- Removing redundant phrasing and consolidating overlapping instructions
- Using structured formats (e.g., JSON schemas and bullet hierarchies) instead of prose explanations where the model responds equally well
- Compressing retrieved context by summarizing or truncating low-relevance passages before including them in the prompt

Output compression targets verbosity in model responses. Chain-of-thought reasoning dramatically improves accuracy on complex tasks, but it also generates thousands of intermediate tokens per request. In production systems where those responses feed into downstream processes rather than human readers, that verbosity often provides no practical value.

Research from Google and Purdue published in April 2026 introduced CROP (Cost-Regularized Optimization of Prompts), an automated prompt optimization method that adds a regularization objective for output length alongside the standard accuracy objective. By penalizing verbose outputs during the prompt optimization phase, CROP forces the meta-optimizer to discover system prompts that elicit concise reasoning. Evaluated on GSM8K, LogiQA, and BIG-Bench Hard, CROP achieved an 80.6 percent reduction in output token consumption while maintaining competitive accuracy. The core insight is practical: you do not need to sacrifice reasoning quality to reduce reasoning length. The model can be guided to produce the same logical steps in far fewer tokens when the prompt is explicitly optimized for that behavior.

For teams not running formal prompt optimization pipelines, a simpler version of this principle applies: brevity is the watchword, so add explicit instructions to your system prompt for tasks where short answers suffice. "Respond in one sentence" or "Return only the JSON object, no explanation" can cut output tokens by 50 to 90 percent on appropriate tasks.

Model Routing: Matching Complexity to Capability

Not every task in a workflow requires the most capable model available. Simple classification, structured extraction from well-formatted input, routing decisions, and validation checks can often be handled by a smaller, faster, cheaper model with negligible quality loss.

Model routing is the practice of dispatching tasks to the appropriate model tier based on estimated complexity. In a well-designed system, a lightweight model handles high-volume routine work while a larger model is reserved for tasks where its reasoning advantage is demonstrably needed.

The economic case is direct. Using Claude Haiku 4.5 at 1 per million input tokens versus Claude Opus 4 at 15 per million input tokens for the same classification task produces a 15x cost reduction. Even routing 60 percent of requests to the cheaper model delivers significant savings across millions of monthly calls.

The practical challenge in model routing is defining routing criteria reliably. Approaches include:

Task type classification

Certain task categories (extraction, formatting, and retrieval ranking) consistently perform well on smaller models; others (multi-step reasoning, code generation, and complex analysis) benefit from larger ones.

Confidence-based escalation

Run a cheap model first and escalate to a larger model only when confidence is low or the output fails validation checks.

Empirical benchmarking by task type

The only reliable way to set routing thresholds is to measure quality degradation on your specific tasks across model tiers. Generic benchmarks are not a substitute.

Exadel Colleague takes this principle seriously by design. As an agent-and-LLM-agnostic platform, Colleague always selects the right model for the task rather than defaulting to the largest available model. For enterprise teams building AI-enabled delivery pipelines, this kind of built-in cost intelligence compounds over thousands of daily agent runs.

Agentic Architecture: Eliminating Structural Waste

The deepest source of token waste in enterprise AI is often not in any individual prompt but in the workflow architecture itself. Agentic systems that invoke LLMs repeatedly for tasks that do not require fresh reasoning are incurring what researchers now call the "Rerun Crisis."

Research published in April 2026 formalized this problem precisely. In a continuous-loop agent, every step of every workflow execution requires an independent LLM inference call. For a 5-step workflow run 500 times, the cost scales as O(M × N) where M is reruns and N is workflow steps. The research paper cited this example: a 5-step workflow over 500 iterations costs approximately 150 under continuous inference, and approximately 15 even with aggressive caching. The proposed solution, agentic compilation, reduces this to under 0.10 by running a single LLM invocation to generate a deterministic JSON workflow blueprint, then executing that blueprint without further model queries.

This compile-and-execute pattern is powerful for deterministic, repetitive workflows. Rather than re-deriving the same action sequence from scratch on each run, the system compiles reasoning into a reusable artifact. The DOM Sanitization Module cited in the paper compresses raw HTML by up to 85percent before the single compilation call, further reducing the token footprint of that initial invocation.

For enterprise teams, agentic architectural principles can make a substantial impact:

→ Separate reasoning from execution

If a workflow's logic is deterministic once derived, invest in compiling it rather than re-running inference on each execution

→ Cache at the workflow level, not just the prompt level

Completed reasoning artifacts, structured plans, and reusable tool call sequences can all be persisted and reused.

→ Audit agent loop counts

Long agentic chains that loop on similar observations are a signal that architectural waste is accumulating. Each unnecessary loop adds tokens at every attached context length.

Meanwhile, at the infrastructure level, KV cache management inside the inference server itself continues to advance. A March 2026 survey from Dell Technologies researchers organizes recent techniques into five categories: cache eviction (selectively dropping low-importance tokens), cache compression (quantization and pooling), hybrid memory (offloading to CPU or NVMe), novel attention mechanisms (linear and log-linear attention variants), and combination strategies. The key finding: no single technique dominates across all deployment scenarios. The optimal approach depends on context length, hardware constraints, and workload characteristics. For teams self-hosting LLMs, this means KV cache optimization should be part of the infrastructure specification from the start, not a retrofit.

Putting It Together: A Prioritization Framework

Token cost optimization has many levers, and knowing which to pull first saves time. Here's a practical sequence for enterprise teams:

Start with prompt caching

It requires minimal architectural change, is supported natively by all major providers, and delivers the largest single reduction for agentic workloads. Cache hits at 10 percent of standard input pricing pay for themselves immediately.

Audit model routing next

Identify which tasks in your workflows are handled by your highest-tier model. For each, ask whether a smaller model can handle it with acceptable quality. Test empirically. Route accordingly.

Compress outputs for programmatic consumers

If your agent's outputs feed a downstream process rather than providing the final output to the user, explicit brevity instructions and prompt optimization for conciseness (the CROP approach) can cut output tokens dramatically.

Revisit architecture for high-frequency workflows

Once the above optimizations are in place, look at whether any repeated workflows can be compiled into deterministic blueprints. This is the highest-effort framework component but potentially provides the highest-reward optimization for production-scale systems.

Monitor continuously

Token consumption is dynamic. New features, longer conversations, and drifting context windows erode optimizations that were once effective. Build token-level observability into your systems from day one.

The Strategic View

Token cost optimization is not just a cost-containment exercise. Done well, it enables teams to run more capable AI at the same budget, to scale agentic systems to higher volumes without linear cost growth, and to build AI-enabled products that remain economically viable as usage grows.

The techniques in this white paper, from prompt caching and output compression to model routing and compile-and-execute architectures, represent the current state of practice across industry and research. The teams that master them are not just saving money. They are building systems that are structurally more efficient, more predictable, and better positioned to absorb the next generation of capabilities as model providers continue to advance.

For organizations evaluating how to integrate these optimizations into real delivery workflows, Exadel Colleague offers a concrete starting point: an autonomous AI delivery teammate built with LLM agnosticism and task-appropriate model selection at its core, designed to deliver measurable ROI from the first sprint.

Frequently Asked Questions

? What is token cost optimization?

Token cost optimization is the practice of reducing the number of tokens processed by an LLM, or reducing their per-token cost, without meaningfully degrading output quality. It spans prompt design, caching strategies, model routing, and agentic workflow architecture. Done well, various studies state it can cut inference spend by 50 to 80 percent in production systems.

? How does prompt caching reduce LLM costs?

Prompt caching stores the key-value (KV) representations computed during the prefill phase so that subsequent requests sharing the same prefix can skip that computation. Cache reads typically cost around 10 percent of standard input pricing, making repeated large system prompts, which are common in agentic applications, dramatically cheaper after the first call.

? When should I use a smaller model instead of a flagship model?

Use a smaller model whenever empirical testing on your specific tasks shows acceptable quality at reduced cost. Classification, structured extraction, routing decisions, and formatting tasks are common candidates. Even routing 60 percent of requests to a cheaper tier can produce significant savings across millions of monthly calls.

? What is the "Rerun Crisis" in agentic AI, and how do I avoid it?

The Rerun Crisis refers to the O(M × N) cost scaling that occurs when every step of every workflow execution triggers a fresh LLM inference call. It can be mitigated through agentic compilation: running a single LLM call to produce a deterministic workflow blueprint, then executing that blueprint without further model queries. This approach can reduce per-workflow inference costs in some cases by 99 percent or more compared to continuous-loop agents.