AN OPULENTBOTS FRAMEWORK

AI TOKEN OPTIMIZATION FRAMEWORK

Cut your AI coding & API costs by up to 70% — copy, paste, save

OPULENTBOTS.COM

OPULENT

CONTENTS

OPULENT

OPULENT BOTS

AI-POWERED BUSINESS AUTOMATION

Every AI coding tool and API bills you the same way: by the token. And almost everyone overpays — resending full histories, paying for the same answer twice, running a premium model on a one-word question.

We deploy AI agents that work around the clock — automating lead capture, customer engagement, and operational workflows. Every solution is built to generate revenue, not just look impressive.

PRIMER

HOW TOKENS ACTUALLY WORK

WHY THIS MATTERS

Every optimization in this guide makes more sense once you know what you're actually paying for. You're not billed by the word or the message — you're billed by the **token**. Understand tokens and the savings become obvious.

WHAT A TOKEN IS

A model doesn't read letters or words — it reads **tokens**: sub-word chunks produced by a tokenizer. In English, **1 token ≈ 4 characters ≈ ¾ of a word**. Common words are one token ("the"); longer or rarer ones split ("optimization" → 2–3 tokens). The tokenizer uses **Byte-Pair Encoding (BPE)** — frequent character sequences get merged into a single token, rare ones get broken up. Practical upshot: **code, JSON, and unusual strings are token-heavy**, so they cost more than plain prose of the same length.

ENCODING: YOUR TEXT → TOKENS (WHAT YOU PAY AS "INPUT")

Before the model sees anything, the tokenizer **encodes** your entire request into a list of integer token IDs — system prompt + rules file + conversation history + any files + your actual question, all concatenated into one long sequence. Every token in that sequence is an **input token**, and you pay for all of them on **every** call.

DECODING: MODEL → TEXT (WHAT YOU PAY AS "OUTPUT")

The model is **autoregressive** — it generates **one token at a time**. To produce each new token it re-reads (attends over) **every previous token**, picks the next one, and **decodes** it back into text. Two consequences: - **Output tokens typically cost ~4–5× input tokens** (varies by provider) — long, chatty answers are expensive. - Generation cost grows with how much context it must attend over each step.

THE ONE THAT SURPRISES PEOPLE: CONTEXT RE-BILLS EVERY TURN

The API is **stateless** — it remembers nothing between calls, so your tool resends the **whole conversation** each time. A 50-message chat sends all 50 messages' tokens again on message 51. That's why long sessions *snowball* — you're re-paying for the entire history on every single turn.

WHY PROMPT CACHING WORKS: THE KV CACHE

Internally, the model computes attention states (**keys & values**) for every token. For a **stable prefix** — your rules, project context, tool definitions — those states are identical every request. **Prompt caching** stores them so the provider skips the recompute and charges a small fraction. That's the ~90% saving on the static prefix: same tokens, but the expensive math is reused, not redone.

THE WHOLE FRAMEWORK, MAPPED TO THE MECHANISM IT ATTACKS

OPTIMIZATION	WHAT IT DOES AT THE TOKEN LEVEL
Context compression (Ch. 2)	Shrinks the input sequence that gets re-sent every turn
Prompt caching (Ch. 3)	Reuses cached KV states for the stable prefix — skips recompute
Model routing (Ch. 4)	Lowers the **price per token** for routine work
Output brevity (Ch. 1)	Fewer expensive **output** tokens
Result caching (Ch. 6)	Avoids re-encoding the same files/searches repeatedly
Session handoff (Ch. 7)	Resets the context length back toward zero

PRO TIP

💡 **See it yourself:** paste the prompt below into your AI tool to make token cost visible — once you can see it, you can cut it.

📋 COPY-PASTE: MAKE YOUR TOKENS VISIBLE

📋 COPY / PASTE

```
Act as a token meter for this session. Tell me:
1. Roughly how many input tokens my last message used (≈ chars ÷ 4).
2. Roughly how many tokens this full conversation is now sending per turn.
3. Which part of the context is largest (history, files, system prompt)?
4. One specific thing I could drop or compress to cut the per-turn token count.
Answer as a short table, and re-run this whenever I say "token check".
```

CHAPTER ONE

THE AGENT RULES FILE (WORKS IN ANY TOOL)

WHY THIS MATTERS

Every modern AI coding tool reads a **rules file** at the start of a session and treats it as standing instructions. Put your token rules there once and every session is leaner — no babysitting. This one block alone typically cuts **20–30%** of wasted tokens by changing how the assistant behaves.

WHERE IT GOES (PICK YOUR TOOL)

- **Claude Code** → `CLAUDE.md` in your project root
- **OpenAI Codex / Cursor / most agents** → `AGENTS.md` (the cross-tool standard) or `.cursorrules`
- **Windsurf** → `.windsurfrules`
- **Gemini CLI** → `GEMINI.md`
- **Direct API** → prepend it to your system prompt

📋 COPY-PASTE: TOKEN RULES (DROP INTO YOUR RULES FILE)

```
## Token Optimization Rules — MANDATORY
```

PRIMER

CONTEXT MANAGEMENT

- NEVER send full conversation history when a summary will do.
- After ~10 messages, summarize older messages before continuing; keep only the last 5 in full detail. Older = key decisions/actions only.
- If a file was read earlier, reference it by name — do NOT re-read unless it changed.
- NEVER dump entire directories into context. Use index/search + selective fetch.

PRIMER

PROMPT EFFICIENCY

- Lead with the answer, not the reasoning. Be concise.
- Don't restate the request or summarize what you just did unless asked.
- If you can say it in 1 sentence, don't use 3.
- Don't add docstrings/comments/types to code you didn't change.

PRIMER

TOOL USAGE

- Read specific file ranges (offset+limit), not whole files.
- Use grep/glob for targeted search instead of reading many files.
- Make surgical edits (find/replace), not full-file rewrites.
- Batch independent tool calls in parallel, not one-by-one.

PRIMER

RESPONSE SIZE

- Default under ~200 tokens unless the task needs more.
- Code changes: show the diff, not the whole file.
- Multi-step tasks: plan in <100 tokens, then execute.

PRIMER

CACHING AWARENESS

- Static project context is cached by the API — don't repeat it.
- Don't re-derive earlier results — reference them. ```

> PRO TIP
>
> 💡 **Cross-tool tip:** keep one canonical `AGENTS.md` and symlink/copy it to `CLAUDE.md`, `.cursorrules`, etc., so every tool obeys the same rules.

CHAPTER TWO

CONTEXT COMPRESSION

WHY THIS MATTERS

After ~15 messages, your conversation history is often **40K–80K tokens** — and the model re-reads *all* of it on every single call, even though it only needs the current task state (~5K tokens). Compressing the history is the single biggest lever: **up to 40–60%** off a

long session.

📋 COPY-PASTE: COMPRESSION PROTOCOL (PASTE WHEN A CHAT GETS LONG)

📋 COPY / PASTE

```
My conversation history is getting long and burning tokens. Apply this now:

1. Take all messages older than the last 5 exchanges.
2. Extract ONLY: decisions made, files changed, errors hit, current task state.
3. Discard: greetings, reasoning chains, repeated info, abandoned attempts.
4. Compress into a single block under 500 tokens, formatted as:

   ## Session Context (compressed)
   - Working on: [task]
   - Files modified: [list]
   - Key decisions: [list]
   - Current blockers: [list]
   - Last action: [what was just done]

5. Use that block as your context going forward instead of the full history.
6. Before each reply, estimate input tokens; if over 30K, compress again.

Apply this now and confirm.
```

> THE POINT
>
> **The Point:** the model gives an equally good answer from 5K tokens of *current state* as from 80K tokens of full history — you're just paying 16× for the difference.

CHAPTER THREE

PROMPT CACHING — EVERY MAJOR PROVIDER

WHY THIS MATTERS

Your static context — rules file, project background, tool definitions — is sent on **every** request but never changes. Prompt caching makes the provider charge a fraction for that repeated chunk. On a big static prefix this is **up to ~90%** off those tokens. Every major provider supports it; the mechanism differs.

THE FRAMEWORK (ONE RULE FOR ALL PROVIDERS)

Put STATIC content first, DYNAMIC content last. Static = rules, project context, tool defs, long documents. Dynamic = the current task + recent messages. Caching only works on a stable prefix, so order matters everywhere.

HOW EACH PROVIDER DOES IT

PROVIDER / TOOL	HOW CACHING WORKS	WHAT YOU DO
Anthropic (Claude API, Claude Code)	Explicit `cache_control: {type:"ephemeral"}` on static blocks; ~5-min TTL. Cache **write ≈1.25×**, **read ≈0.1×** input cost.	Claude Code caches your system prompt + rules file automatically. On the API, mark static blocks.
OpenAI (GPT-4.1 / o-series, Codex)	**Automatic** prompt caching for prompts ≥1,024 tokens; **~50%+ discount** on cached input. No code change.	Just keep the static prefix identical and first.
Google Gemini	**Context caching** — upload large/static context once as `CachedContent`, reference it by handle; reduced token rate + small storage fee.	Cache big docs/system context, reuse the handle.
Most others (xAI, DeepSeek, etc.)	Increasingly automatic prefix caching at a discount.	Same rule: static prefix first, keep it stable.

📋 COPY-PASTE: CACHING SETUP (PASTE TO YOUR ASSISTANT / API HELPER)

📋 COPY / PASTE

```
Enable prompt caching for all static, repeated content.

1. Split the system prompt into STATIC (rules, project context, tool defs) and
   DYNAMIC (current task, recent messages).
2. Put STATIC first and keep it byte-identical across requests.
3. Anthropic API: add cache_control {type:"ephemeral"} to the static blocks.
   OpenAI: no code needed — caching is automatic on a stable >1k-token prefix.
   Gemini: store the static context as CachedContent and reference it.
4. Never interleave dynamic text inside the static prefix (it breaks the cache).

Confirm what is now cached and the expected read-vs-write cost.
```

CHAPTER FOUR

SMART MODEL ROUTING — ACROSS PROVIDERS

WHY THIS MATTERS

Most people run their most expensive model for *everything* — including "what's the current git branch." That 3-token answer costs the same input processing as a complex refactor. Routing routine work to a cheap model saves **up to 15–25%** (often more) with zero quality loss.

THE TIERS (ROUTE BY TASK DIFFICULTY, NOT HABIT)

TIER	MODELS (EXAMPLES)	USE FOR
Premium (deep reasoning)	Claude Opus · OpenAI o-series · Gemini 2.5 Pro	New code, architecture, hard debugging, multi-step tool chains
Mid (daily driver)	Claude Sonnet · GPT-4.1 · Gemini 2.5 Flash	Most coding and analysis
Cheap (routine)	Claude Haiku · GPT-4.1 mini · Gemini 2.5 Flash-Lite	Summaries, formatting, boilerplate, git ops, reading test output, status checks

> **NOTE**
> The cheap tier is often **15–100× cheaper** per token than the premium tier. Routing 40–60% of tasks there is the easiest money in this guide.

HOW TO SWITCH IN EACH TOOL

- **Claude Code:** `/model` to change model mid-session (use cheap for simple turns).
- **OpenAI Codex / Cursor / Windsurf:** pick the model in the model selector or config; set a cheaper default and bump up only when needed.
- **Direct API:** set the `model` parameter per call based on task type.

📋 COPY-PASTE: ROUTING RULE (PASTE INTO YOUR RULES FILE)

📋 COPY / PASTE

```
Route by difficulty, not habit:
- CHEAP model: summaries, formatting, boilerplate, git/file ops, reading output,
  status checks, simple factual answers.
- MID model: most coding and analysis.
- PREMIUM model: new code, architecture, complex debugging, multi-step reasoning.
Default to the cheaper tier and escalate only when the task clearly needs it.
```

CHAPTER FIVE

TOKEN BUDGET ENFORCEMENT

WHY THIS MATTERS

Without limits, sessions silently balloon — a 2-hour agent session can hit 500K–1M tokens because every message drags the growing history. Hard budgets force compression at the right moments and stop the snowball.

📋 COPY-PASTE: BUDGET RULES (PASTE AT SESSION START)

📋 COPY / PASTE

```
Apply hard budget rules this session:

Per request:  cap input context at ~32K tokens; cap output at ~4K unless asked.
              If input would exceed 32K, compress history first (Chapter 2).
Per session:  track cumulative tokens.
              ~100K → warn me + suggest compressing.
              ~200K → mandatory compression before continuing.
              ~500K → suggest a fresh session with a handoff summary.
Before big ops (reading 10+ files, long generation): estimate tokens (~$cost)
              and ask before anything over ~20K tokens.
Triggers:     after reading >5 files, summarize — don't keep raw content.
              after ~20 tool calls, compress to current state.
              after an error/retry cycle, drop the failed attempts, keep the fix.

Start tracking now and tell me the current estimated session token count.
```

CHAPTER SIX

RESULT CACHING (STOP PAYING TWICE)

WHY THIS MATTERS

In a typical session the same 5–10 files get read 3–4 times each as context gets lost, and the same code gets re-analyzed repeatedly. Each re-read is 1–5K tokens wasted. Caching *results in-context* eliminates one of the biggest hidden costs.

📋 COPY-PASTE: RESULT-CACHING RULE (PASTE INTO YOUR RULES FILE)

📋 COPY / PASTE

```
Cache results in-context; don't recompute:
- After reading a file, note its key contents in 1–2 sentences and reference
  that later ("as seen in src/app.ts") instead of re-reading.
- Store grep/glob findings and analysis conclusions; reuse them.
- Read all files for a multi-file task FIRST, summarize, then work from the summary.

Always fetch fresh (never cache): a file after you edited it, git status after a
commit, build/test output after a code change, or anything I say "check again".
```

CHAPTER SEVEN

STRATEGIC SESSION MANAGEMENT

WHY THIS MATTERS

The biggest single cost in AI coding is **session length**. A long 3-hour session with full history can run $50–$100 in a worst case; the same work in three focused 1-hour sessions with handoffs costs a fraction, because each starts with clean, compressed context. **Up to 50–70%** on long work.

THE PLAY

Compact context **between natural phases** (research→plan→build→test→debug→fix) and start a fresh session after a major feature, a task switch, or ~2 hours.

📋 COPY-PASTE: PHASE HANDOFF (USE BETWEEN PHASES OR TO START A FRESH SESSION)

```

---

PRIMER

# PHASE HANDOFF

Completed: [what was just done] Result: [outcome] Current state: [what exists now] Next: [what needs to happen] Key files: [list of modified files] Open issues: [anything unresolved]

To resume in a NEW session, start with only: "Continuing work on [project]. Here's where I left off: [paste handoff]. Start here." ```
```

THE POINT

The Point: a handoff gives the model perfect context in ~200 tokens instead of dragging 200K tokens of dead history into the new session.

CHAPTER EIGHT

THE WEEKLY TOKEN AUDIT

WHY THIS MATTERS

What gets measured gets cut. Five minutes a week keeps the savings from quietly creeping back.

📋 COPY-PASTE: TOKEN AUDIT (PASTE WEEKLY)

📋 COPY / PASTE

```
Run a token audit for this session and answer as a short table:
1. How many messages have we exchanged?
2. Estimate total input and output tokens.
3. How many files were read? Were any read more than once?
4. How many tool calls were made? How many could have been parallel?
5. Estimated cost of this session so far?
6. The single biggest token waste you can identify?
7. One specific change to cut cost for the rest of this session.
Be honest about waste.
```

BONUS

THE ALL-IN-ONE PROMPT

Don't want to apply the chapters one at a time? Paste this **single block** into your AI agent (Claude Code, Codex, Cursor, Gemini CLI, or any chat) — or into your rules file — and it implements the entire framework at once.

📋 COPY-PASTE: THE COMPLETE FRAMEWORK (ONE PROMPT)

📋 COPY / PASTE

```
You are now optimized for minimum token cost. Adopt ALL of these rules for the
rest of our work and confirm you've applied them:

1. CONTEXT — Never resend full history. After ~10 messages, keep the last 5 in
   detail and compress everything older to: decisions, files changed, errors,
   current state. Never dump whole directories — search + fetch only what's needed.
   If you read a file already, reference it by name; don't re-read unless it changed.

2. BREVITY — Lead with the answer. Don't restate my request or narrate what you
   did. Show diffs, not whole files. No comments/docstrings on code you didn't change.

3. TOOLS — Read file ranges (offset+limit), not whole files. Use grep/glob to find
   things. Make surgical find/replace edits. Batch independent tool calls in parallel.

4. CACHING — Treat static context (these rules, project background, tool defs) as a
   stable prefix that stays first and unchanged so the provider can cache it. Don't
   repeat static context; don't recompute earlier results — reference them.

5. MODEL ROUTING — Assume a cheap model for routine work (summaries, formatting,
   boilerplate, git/file ops, reading output, status checks) and a premium model
   only for new code, architecture, and hard debugging. Tell me when a task should
   switch tiers.

6. BUDGET — Cap input context at ~32K tokens and output at ~4K unless I ask for more.
   Warn me at ~100K cumulative session tokens, compress at ~200K, suggest a fresh
   session at ~500K. Before any big operation (10+ files, long generation), estimate
   the token cost and ask before exceeding ~20K tokens.

7. RESULT CACHING — After reading files or running searches/analysis, store the
   findings in 1–2 lines and reuse them instead of repeating the work. Always re-fetch
   only: files you edited, git status after commits, build/test output after changes.

8. SESSIONS — At natural phase breaks (research→plan→build→test→debug→fix), compress
   to a short handoff (completed / result / current state / next / key files / open
   issues). Recommend a fresh session after a major feature or ~2 hours.

Confirm: list which rules are now active and give me an estimated % token reduction
for how we'll work from here.
```

PRO TIP

💡 **Best practice:** paste this once into your rules file (`CLAUDE.md` / `AGENTS.md` / `.cursorrules`) so every future session starts optimized — no re-pasting.

BONUS

QUICK-REFERENCE CARD + THE MATH

QUICK REFERENCE

TECHNIQUE	UP-TO SAVINGS	EFFORT	WHEN
Rules file (Ch. 1)	20–30%	Paste once	Every project
Context compression (Ch. 2)	40–60%	Paste when long	After ~15 msgs
Prompt caching (Ch. 3)	~90% on static prefix	One-time setup	Always
Model routing (Ch. 4)	15–25%+	Switch per task	Every session
Budget enforcement (Ch. 5)	Prevents blow-ups	Paste at start	Every session
Result caching (Ch. 6)	10–15%	Paste once	Every project
Session management (Ch. 7)	50–70% on long work	Split at phase breaks	Sessions > 2 hrs

Stacked, these typically land a 50–70% reduction on a cache- and routing-heavy workload — short one-off prompts save less.

THE MATH (EXAMPLE)

A heavy AI-coding user at ~$30–$50/day, after stacking these on a favorable workload, often runs ~$12–$20/day for the same work — roughly **$540–$900/month** saved. Your mileage depends on workload; measure your own before and after.

PRIMER

YOU CUT THE BILL. NOW SCALE THE BUSINESS.

Cutting AI cost is defense. The real upside is turning AI from a tool you *pay for* into a system that *runs your operation and produces revenue*.

We mapped that path — from "AI answers questions" to "AI runs a P&L" — into a free 2-minute diagnostic that tells you your exact stage and your next move.

→ **Take the AI Maturity Diagnostic:** opulentbots.com/matrix

At the end you can book a **free AI Cost Audit** that runs on your real numbers and hands you a measured savings floor.

READY TO IMPLEMENT?

You can run every play in this guide yourself. If you'd rather we build and run it **for** you — the AI phone agent, the automations, the campaigns, integrated with your existing tools — book a free strategy call.

BOOK A FREE STRATEGY CALL

https://opulentbots.com