CC STRATEGIC | CLAUDE CODE PLAYBOOK

WORKSPACE IS THE PRODUCT

Build a Claude Code statusline that shows what's actually eating your context.

3-step setup. One file. Zero dependencies.

By Charles J Dove

C&C Strategic Consulting | ccstrategic.io

```
[CAVEMAN]   @yourtag   ctx:33% (326k)
cached:323k new:2.9k
skl:7.9k   mcp:1.3k   hk:881
```

01 / WHY THIS MATTERS

Claude Code is the application. Your workspace is the OS.

If your operating system is garbage, no model saves you. Better prompts won't fix it. A bigger model won't fix it either.

The .claude/ folder IS the product Claude reads. Every file, every rule, every hook. That is your context. That is what the model is actually reading every single turn.

Most people configure 1 folder. There are 3. And they have no idea what is loaded into context until they hit a wall, watch their cache break, or burn through tokens for no apparent reason.

> *"The model does not get smarter. Your environment does."*

This playbook hands you a single Python file that drops into your Claude Code config and shows you, in real time, exactly which parts of your workspace are eating context. Then it shows you how to read those numbers so you can fix what is heavy.

02 / WHAT YOU'LL SEE

Your statusline, decoded.

Three lines at the bottom of every Claude Code session. Each line tells you something different about what your session is costing you.

```
[CAVEMAN]    @yourtag    ctx:33% (326k)
cached:323k  new:2.9k
skl:7.9k     mcp:1.3k     hk:881
```

[CAVEMAN]	Optional plugin badge. Skip if you don't use the caveman plugin.
@yourtag	Your handle. Only shows when cwd is inside a repo you tagged.
ctx:N% (Nk)	Total context window usage. Color shifts gray to yellow to orange to red.
cached:Nk	Tokens served from prompt cache this turn. Cheap (~10% of normal price).
new:Nk	New content this turn. Full price. When this spikes, your cache broke.
skl / mcp / hk	Token cost from each context source. Skills, MCP servers, hook outputs.

03 / INSTALL

Install in 3 steps.

No pip install. No env vars. No node_modules. Three steps, one paste each.

STEP 1 / Save the script

Open Finder, press **Cmd+Shift+G**, paste `~/.claude/hooks/`. If the hooks folder doesn't exist, create it. Save the script from pages 5-6 as `claude-statusline.py` inside that folder.

STEP 2 / Wire it into settings.json

Open `~/.claude/settings.json` in any editor. Add this block at the top level (alongside your existing keys):

```
"statusLine": {
  "type": "command",
  "command": "python3 /Users/YOUR_USERNAME/.claude/hooks/claude-statusline.py"
}
```

Replace **YOUR_USERNAME** with your macOS username. Find it by running `whoami` in Terminal.

STEP 3 / Restart Claude Code

Quit Claude Code (**Cmd+Q**), reopen it, start any new session. The statusline appears at the bottom. Numbers populate as the session runs.

04 / THE SCRIPT

Paste this into claude-statusline.py

~265 lines, Python stdlib only. Three customization points marked below with **CHANGE THIS** comments. Then jump to the next page if it continues.

```
#!/usr/bin/env python3
"""
Statusline wrapper.
Combines: caveman badge + @charlieautomates tag + context % + component counts.
Reads JSON on stdin from Claude Code.
"""
import hashlib
import json
import os
import re
import subprocess
import sys
from pathlib import Path

CAVEMAN_SCRIPT = "/Users/user/.claude/plugins/cache/caveman/caveman/84cc3c14fa1e/hooks/cavem..."
CHARLIE_REPO_MARKER = "Charlieautomates"
CACHE_DIR = "/tmp/claude-statusline-cache"


def caveman_badge() -> str:
    if not os.path.exists(CAVEMAN_SCRIPT):
        return ""
    try:
        r = subprocess.run(
            ["bash", CAVEMAN_SCRIPT],
            capture_output=True,
            text=True,
            timeout=2,
        )
        return r.stdout
    except Exception:
        return ""


def charlie_tag(cwd: str) -> str:
    if CHARLIE_REPO_MARKER in cwd:
        return " \033[38;5;141m@charlieautomates\033[0m"
    return ""


def last_usage(transcript_path: str):
    if not transcript_path or not os.path.exists(transcript_path):
        return None
    last = None
    try:
        with open(transcript_path, "r", encoding="utf-8", errors="ignore") as f:
            for line in f:
                try:
                    msg = json.loads(line)
                except Exception:
                    continue
                if msg.get("type") != "assistant":
                    continue
                u = msg.get("message", {}).get("usage") or msg.get("usage")
                if u:
                    last = u
    except Exception:
        return None
    return last


def _usage_breakdown(transcript_path: str, model_id: str):
    """Return (pct, total, cached, new) or None."""
    u = last_usage(transcript_path)
    if not u:
```

Customize: change `CHARLIE_REPO_MARKER` to your repo folder name and the `@charlieautomates` string in `charlie_tag()` to your own handle. Don't use the caveman plugin? Delete `caveman_badge()` and its call in `main()`.

```
        return None
    budget = 1_000_000 if "[1m]" in model_id else 200_000
    cache_read = int(u.get("cache_read_input_tokens", 0))
    cache_creation = int(u.get("cache_creation_input_tokens", 0))
    raw_input = int(u.get("input_tokens", 0))
    cached = cache_read
    new = raw_input + cache_creation
    total = cached + new
    pct = round(100 * total / budget) if budget else 0
    return pct, total, cached, new


def ctx_only(transcript_path: str, model_id: str) -> str:
    b = _usage_breakdown(transcript_path, model_id)
    if not b:
        return ""
    pct, total, _cached, _new = b
    color = "\033[38;5;245m"
    if pct >= 80:
        color = "\033[38;5;196m"
    elif pct >= 60:
        color = "\033[38;5;208m"
    elif pct >= 40:
        color = "\033[38;5;220m"
    reset = "\033[0m"
    k = total / 1000
    return f" {color}ctx:{pct}% ({k:.0f}k){reset}"


def cache_split(transcript_path: str, model_id: str) -> str:
    b = _usage_breakdown(transcript_path, model_id)
    if not b:
        return ""
    _pct, _total, cached, new = b
    dim = "\033[38;5;240m"
    reset = "\033[0m"
    cached_k = cached / 1000
    new_k = new / 1000
    return f"{dim}cached:{cached_k:.0f}k new:{new_k:.1f}k{reset}"


def context_pct(transcript_path: str, model_id: str) -> str:
    """Legacy single-line combined output (kept for backwards compat)."""
    ctx = ctx_only(transcript_path, model_id)
    cs = cache_split(transcript_path, model_id)
    if not ctx:
        return ""
    return f"{ctx} {cs}" if cs else ctx


def find_skill_md(name: str, cwd: str):
    """Find SKILL.md for an invoked skill. Returns Path or None."""
    candidates = [
        Path.home() / ".claude" / "skills" / name / "SKILL.md",
    ]
    if cwd:
        candidates.append(Path(cwd) / ".claude" / "skills" / name / "SKILL.md")
    # Also support namespaced skills like "carl:manager" -> carl/manager/SKILL.md
    if ":" in name:
        ns, sub = name.split(":", 1)
        candidates.append(Path.home() / ".claude" / "plugins" / "cache" / ns / ns / sub / "S...
    for p in candidates:
        if p.exists():
            return p
    return None


def parse_token_usage(transcript_path: str, cwd: str) -> dict:
    """Estimate token cost (chars/4) per category from transcript:
    - skl: bytes of SKILL.md files for skills actually invoked this session
    - mcp: bytes of MCP tool schema blocks loaded into context
    - hk: bytes of hook output blocks (system_reminder hook fires)
    Cached by transcript size+mtime."""
    empty = {"skl": 0, "mcp": 0, "hk": 0}
```

```
    if not transcript_path or not os.path.exists(transcript_path):
        return empty
    try:
        st = os.stat(transcript_path)
    except OSError:
        return empty

    os.makedirs(CACHE_DIR, exist_ok=True)
    key = hashlib.md5(transcript_path.encode()).hexdigest()[:12]
    cache_file = Path(CACHE_DIR) / f"{key}.json"
    if cache_file.exists():
        try:
            cached = json.loads(cache_file.read_text())
            if cached.get("size") == st.st_size and cached.get("mtime") == st.st_mtime:
                return {"skl": cached.get("skl", 0), "mcp": cached.get("mcp", 0), "hk": cach...
        except Exception:
            pass

    skill_chars = 0
    mcp_chars = 0
    hook_chars = 0
    invoked_skills = set()

    skill_launch_re = re.compile(r"Launching skill:\s*([a-zA-Z][a-zA-Z0-9:_-]+)")

    try:
        with open(transcript_path, "r", encoding="utf-8", errors="ignore") as f:
            for line in f:
                try:
                    msg = json.loads(line)
                except Exception:
                    continue

                # Walk attachments — Claude Code stores hook fires, MCP defs, and skill list...
                att = msg.get("attachment") or {}
                if att:
                    att_type = att.get("type", "")
                    content = att.get("content") or att.get("snippet") or ""
                    content_len = len(content) if isinstance(content, str) else 0
                    # MCP attachments use addedLines/addedBlocks lists, not content field
                    if att_type in ("deferred_tools_delta", "mcp_instructions_delta"):
                        for field in ("addedLines", "addedBlocks", "addedNames"):
                            for s in att.get(field, []) or []:
                                if isinstance(s, str):
                                    mcp_chars += len(s)
                    elif att_type in ("hook_success", "hook_additional_context"):
                        hook_chars += content_len
                    elif att_type == "skill_listing":
                        skill_chars += content_len

                # Skill invocations: track unique names launched via Skill tool
                # Search assistant message text for "Launching skill:" markers
                blob = json.dumps(msg)
                for m in skill_launch_re.finditer(blob):
                    name = m.group(1)
                    if name in invoked_skills:
                        continue
                    invoked_skills.add(name)
                    p = find_skill_md(name, cwd)
                    if p:
                        try:
                            skill_chars += p.stat().st_size
                        except OSError:
                            pass
    except Exception:
        pass

    result = {
        "size": st.st_size,
        "mtime": st.st_mtime,
        "skl": skill_chars // 4,
        "mcp": mcp_chars // 4,
        "hk": hook_chars // 4,
    }
```

```
    try:
        cache_file.write_text(json.dumps(result))
    except Exception:
        pass
    return {"skl": result["skl"], "mcp": result["mcp"]
, "hk": result["hk"]}


def fmt_k(tokens: int) -> str:
    if tokens < 1000:
        return f"{tokens}"
    return f"{tokens / 1000:.1f}k"


def components_badge(transcript_path: str, cwd: str) -> str:
    usage = parse_token_usage(transcript_path, cwd)
    skl = usage.get("skl", 0)
    mcp = usage.get("mcp", 0)
    hk = usage.get("hk", 0)
    if skl == 0 and mcp == 0 and hk == 0:
        return ""
    dim = "\033[38;5;240m"
    skill_c = "\033[38;5;115m"  # mint
    mcp_c = "\033[38;5;141m"    # purple
    hook_c = "\033[38;5;180m"   # gold
    reset = "\033[0m"
    return f"{dim} | {skill_c}skl:{fmt_k(skl)}{reset} {mcp_c}mcp:{fmt_k(mcp)}{reset} {hook_c...


def main() -> None:
    try:
        data = json.load(sys.stdin)
    except Exception:
        data = {}
    cwd = data.get("cwd", "")
    transcript_path = data.get("transcript_path", "")
    model_id = data.get("model", {}).get("id", "")

    line1 = (caveman_badge() or "") + charlie_tag(cwd) + ctx_only(transcript_path, model_id)
    line2 = cache_split(transcript_path, model_id)
    # components_badge() returns " | skl:... mcp:... hk:..." with a leading
    # " | " separator meant for horizontal joining. Strip it for line 3.
    raw_components = components_badge(transcript_path, cwd)
    line3 = re.sub(r"^(\s*\033\[[0-9;]*m)?\s*\|\s*", r"\1", raw_components) if raw_component...

    output = "\n".join(p for p in (line1, line2, line3) if p)
    sys.stdout.write(output)


if __name__ == "__main__":
    main()
```

05 / READING THE NUMBERS

What each number tells you.

Each number maps to a real source of token cost. When one category gets heavy, you know exactly what to trim.

TOTAL CONTEXT

`ctx:N%`
Your context window fill. Color escalates: gray under 40%, yellow 40-60%, orange 60-80%, red 80%+. Plan a /clear before red.

`cached:Nk`
Tokens served from Anthropic's prompt cache this turn. About 10% of normal input price. Stays cached for 5 minutes after last touch.

`new:Nk`
New content this turn. Full price. Includes your latest message plus any new content being cached for first time. When this spikes, the cache broke.

WHAT IS LOADED

`skl:Nk`
Tokens from your skill index plus any SKILL.md bodies you invoked this session. Heavy means too many skills available or a bloated skill body.

`mcp:Nk`
Tokens from MCP server instructions and tool name listings. Each MCP server adds 200-500 tokens just by being registered.

`hk:Nk`
Cumulative output from every hook fire. Hooks that inject big context blocks every UserPromptSubmit add up fast.

> **Heavy in one category?** Trim that source. Skills you never use? Delete them. MCPs you don't need? Disable them. Hooks that fire too often or output too much? Refactor.

06 / NEXT

Now go architect your workspace.

This statusline shows you what your context actually contains.

The next step is fixing what's heavy. That's where most of the leverage hides. Skills you never invoke. MCPs you registered and forgot. Hooks that fire on every prompt and shouldn't.

JOIN CC STRATEGIC AI ON SKOOL ->

Where we build Claude Code workspaces that actually work.

skool.com/cc-strategic-ai/about

RUN A FREE BOTTLENECK FINDER ->

Find what's actually killing your weekly output. 5 minutes.

charlieautomates.com/diagnose

Charles J Dove / C&C Strategic Consulting / ccstrategic.io