Context Priming: Proactive Context Synthesis for Coding Agents

Boris Djordjevic

199 Biotechnologies
February 2026 — Version 2.1

Abstract. Large Language Model (LLM) coding agents face a fundamental architectural bottleneck: context management. Current approaches are overwhelmingly *reactive* — compressing, truncating, or summarizing context only when the window fills up. This paper proposes **Context Priming**, a paradigm shift where coding agents *proactively construct* their optimal context before beginning any task. Rather than subtracting information from an overflowing window, the agent synthesizes a purpose-built context from multiple sources: accumulated memories, codebase structure, flagged priorities, past mistakes, and an explicit outcome hierarchy. This "soft compaction" operates as generative context construction — the inverse of the reductive compression that dominates the field today. We formalize the priming objective, present a reference implementation with measured performance, survey existing approaches, and identify the specific gap Context Priming fills.

---

1 Introduction

1.1 The Context Crisis

The effectiveness of LLM-based coding agents is fundamentally constrained by what fits in their context window. Despite context windows expanding from 4K to 200K+ tokens between 2023 and 2026, the core problem persists: *having more space doesn't mean you know what to put in it*.

Current coding agents — Claude Code, GitHub Copilot, Cursor, Windsurf, and others — face a recurring failure mode. They begin a task, accumulate context through file reads, tool calls, and conversation turns, then degrade as the context fills with irrelevant artifacts from earlier exploration. Studies have documented "context rot" — the phenomenon where model accuracy on information retrieval decreases as context length increases, even when the information is present in the window.[1]

We can express this degradation formally. Let $A(t)$ denote agent accuracy at context utilization fraction $t \in [0, 1]$. Empirical measurements show:

$$A(t) \approx A_0 \cdot \left(1 - \alpha t^{\beta}\right) \tag{1}$$

where A_0 is baseline accuracy, $\alpha \in [0.3, 0.5]$ is the degradation coefficient, and $\beta \approx 1.5$ reflects the super-linear decay observed in practice. At 80% context utilization ($t = 0.8$), agents typically lose 20–35% of their retrieval accuracy.

1.2 The Reactive Paradigm

The industry's response has been reactive context management:

- **Auto-compaction** triggers when context reaches a threshold (e.g., Claude Code compacts at 95% capacity)
- **Truncation** drops older messages beyond a sliding window
- **Recursive summarization** compresses conversation history into progressively shorter summaries
- **RAG (Retrieval-Augmented Generation)** fetches relevant documents on demand

All of these approaches share a common assumption: context is something that *accumulates and must be managed*. They are subtractive — their goal is to remove or compress what's already there.

1.3 The Priming Alternative

We propose inverting this assumption entirely. Instead of asking *"what should we remove?"*, Context Priming asks *"what is the optimal context to construct for this specific task?"*

A primed agent doesn't start with an empty context and reactively accumulate information. It doesn't start with a full context and reactively compress it. Instead, it begins by *synthesizing* a purpose-built context — drawing from memories, codebase knowledge, past lessons, and goal awareness — and then operates within that optimized frame.

2 The Problem Space

2.1 Why Context Matters More Than Model Capability

Anthropic's engineering team has stated that "the single most important determinant of AI agent effectiveness is providing the right context."[1] The field has converged on this insight: prompt engineering is about *what you say*; context engineering is about *what the model sees*. And what the model sees matters more.

Every token in context costs attention. Let n be the number of tokens and d the model dimension. The self-attention cost scales as:

$$\mathcal{O}(n^2 \cdot d) \tag{2}$$

Larger contexts don't just consume memory — they dilute the model's focus. Research from Voltropy (2026) on Lossless Context Management demonstrated that agents allowed to manage their own memory frequently degrade their own performance through poor prioritization.[2]

2.2 The Five Failure Modes

Current coding agents exhibit five recurring failure modes related to context:

1. **Cold-start blindness.** New sessions begin with no relevant context, forcing the agent to rediscover project conventions, architecture, and past decisions from scratch.
2. **Memory bloat.** Accumulated memories and lessons grow into massive files that cannot fit in context, forcing either truncation (losing valuable lessons) or no memory at all.
3. **Goal drift.** Without explicit awareness of the outcome hierarchy, agents optimize for the literal user request rather than the actual desired outcome. A request to "fix this test" may require understanding that the test exists to validate a migration that serves a larger architecture change.
4. **Lesson amnesia.** Past mistakes and their solutions are recorded but never surfaced at the right moment. A memory about "always use absolute imports in this project" is useless if it's buried on line 847 of a memory file when the agent is writing a new module.
5. **Context pollution.** Exploratory tool calls (file reads, searches, failed approaches) accumulate in context and dilute the signal-to-noise ratio for the actual task.

2.3 The Cost of Getting Context Wrong

ContextBench (2026), a benchmark for context retrieval in coding agents, demonstrated that even state-of-the-art agents retrieve the wrong context for coding tasks 30–40% of the time. This isn't a model capability issue — it's an architectural one. The agents have access to the right information; they fail to surface it at the right time.

We define *context precision* as the fraction of primed tokens that are actually relevant:

$$P_{\text{context}} = \frac{|C_{\text{relevant}} \cap C_{\text{primed}}|}{|C_{\text{primed}}|} \tag{3}$$

Current approaches achieve $P_{\text{context}} \approx 0.4$–$0.6$. Context Priming targets $P_{\text{context}} \geq 0.8$ by applying task-specific relevance scoring before context construction.

3 Existing Approaches and Their Limitations

3.1 Reactive Compaction

Examples: Claude Code auto-compact, Google ADK context compression, Forge Code compaction.

Approach: Monitor context usage, trigger summarization when approaching limits.

Limitation: By the time compaction fires, context is already degraded. Compaction is lossy — it cannot recover information that was never loaded, and it inevitably discards nuances that may become relevant later. The information loss $\mathcal{L}$ of reactive compaction can be modeled as:

$$\mathcal{L}_{\text{reactive}} = H\big(C_{\text{original}}\big) - H\big(C_{\text{compacted}}\big) \geq 0 \tag{4}$$

where H denotes the task-relevant information entropy. The inequality holds strictly because compaction is a lossy operation that cannot distinguish between task-essential and exploratory context.

3.2 Memory Systems

Examples: Letta/MemGPT (hierarchical memory), SimpleMem (semantic compression), `MEMORY.md` files.

Approach: Store agent experiences in structured memory systems with retrieval mechanisms.

Limitation: Memory systems solve *storage* but not *synthesis*. SimpleMem achieves impressive compression (30× token reduction) but still relies on query-time retrieval — the agent must know what to ask for.[4] Memory files like `MEMORY.md` provide persistence but are loaded wholesale, with no task-specific curation. Letta's hierarchical memory (core, archival, recall) is architecturally sophisticated but optimizes for long-running agent sessions, not for pre-task preparation.[5]

3.3 Static Context Loading

Examples: elizaOS `/context-prime`, `CLAUDE.md` files, `AGENTS.md` conventions.

Approach: Load predefined context files (README, project structure, conventions) at session start.

Limitation: Static context is one-size-fits-all. A debugging task and a feature implementation task in the same project need fundamentally different context. Recent research (Koylan, 2026) found that LLM-generated general context actually *decreases* performance by -3% on average on AGENTBENCH, while developer-written context only marginally improves it ($+4\%$).

3.4 Planning-First Approaches

Examples: ContextKit (4-phase methodology), CodePlan (pseudocode planning).[11]

Approach: Structure development into explicit planning phases before implementation.

Limitation: Planning-first approaches improve *workflow* but not *context quality*. They tell the agent *how* to work but don't optimize *what the agent sees*. A planning phase that runs inside a polluted context will produce a polluted plan.

3.5 Agentic Context Engineering (ACE)

Reference: "Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models" (ICLR 2026).[3]

Approach: Treats contexts as evolving playbooks that accumulate, refine, and organize strategies through generation, reflection, and curation. Achieves $+10.6\%$ improvement on agent benchmarks.

Limitation: ACE focuses on *evolving system prompts over time* — it's about the gradual improvement of standing context through experience. It doesn't address the per-task synthesis problem: given a specific task right now, what is the optimal context to construct from all available sources? ACE evolves the playbook; Context Priming constructs the play.

4 The Context Priming Proposal

4.1 Core Concept

Context Priming is the process by which a coding agent, upon receiving a task, *proactively constructs its optimal starting context* before executing any work. It is:

- **Proactive**, not reactive — it happens before work begins, not when context fills up
- **Constructive**, not reductive — it builds the right context rather than compressing the wrong one
- **Task-specific** — different tasks produce different primed contexts
- **Multi-source** — it synthesizes from memories, codebase, goals, priorities, and external knowledge
- **Goal-aware** — it considers the outcome hierarchy, not just the literal task

4.2 The Priming Objective

We formalize the priming objective as an optimization problem. Given a task τ, a set of available sources $\mathcal{S} = \{s_1, s_2, ..., s_n\}$, and a context budget B (in tokens), find the optimal subset $\mathcal{S}^* \subseteq \mathcal{S}$ that maximizes task-relevant information within the budget:

$$\mathcal{S}^* = \arg\max_{\mathcal{S}' \subseteq \mathcal{S}} \sum_{s_i \in \mathcal{S}'} R(s_i, \tau) \quad \text{subject to} \quad \sum_{s_i \in \mathcal{S}'} |s_i| \leq B \tag{5}$$

where $R(s_i, \tau) \in [0, 1]$ is the relevance score of source s_i to task τ, and $|s_i|$ denotes the token count of source s_i.

The context budget B is defined as a fraction of the platform's available coding context:

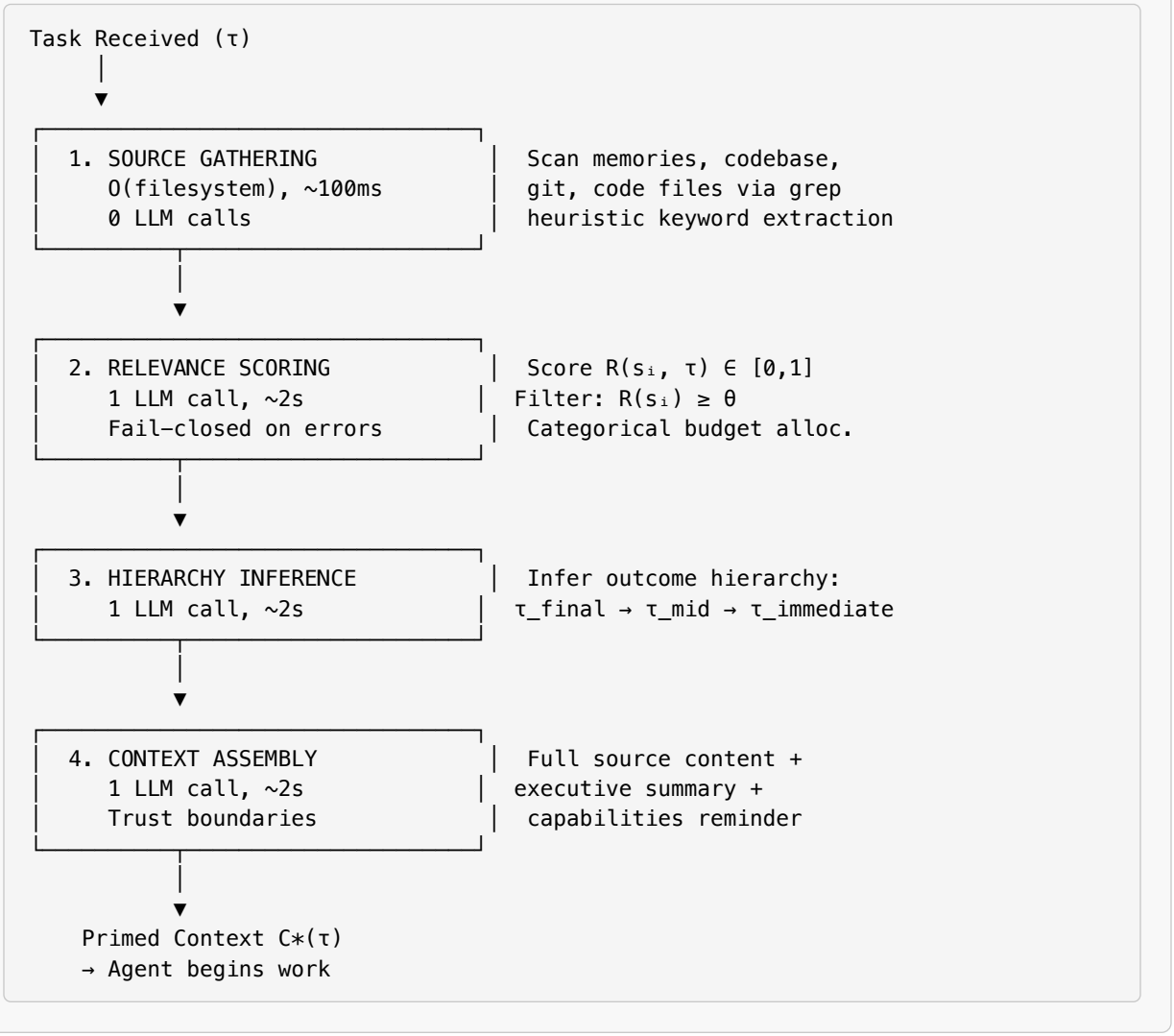
$$B = \gamma \cdot C_{\text{platform}} \tag{6}$$

where $\gamma \in [0.1, 0.4]$ is the budget fraction (default $\gamma = 0.25$) and C_{platform} is the platform's usable context window. Table 1 shows the platform-specific values.

Platform	C_{platform}	B (at $\gamma = 0.25$)
Claude Code	120k	30k
Claude API (raw)	200k	50k
Gemini CLI	1,000k	250k
Codex CLI	200k	50k
OpenCode	128k	32k

Table 1: Platform context budgets. C_{platform} is the usable coding context; B is the priming budget at $\gamma = 0.25$.

4.3 The Priming Pipeline



Listing 1: The Context Priming pipeline. Three LLM calls using a fast model (e.g., Sonnet) plus one filesystem scan. Total overhead: 5–8 seconds.

4.4 Source Categories

Context Priming draws from six source categories, each contributing different signal:

Source Category	Signal	Example
Accumulated Memories	Past lessons, patterns, mistakes	*"This project's API always requires auth headers"*
Codebase Structure	Architecture, conventions, deps	Module layout, import patterns, test structure
Source Code Files	Actual implementation code	Files matching task keywords via `grep`
Flagged Priorities	What the team considers important	*"Performance is critical — always benchmark"*
Recent History	What was recently changed and why	Last 10 commits, staged/unstaged changes
External Knowledge	Documentation, research, standards	API docs, framework best practices

Table 2: The six source categories gathered by Context Priming.

4.5 Outcome Hierarchy

A key innovation of Context Priming is explicit outcome awareness. Most agents treat each task atomically — "fix this bug" or "add this feature." But tasks exist in hierarchies:

τ_{final}: Ship the v2 platform by Q2
↓
τ_{mid}: Complete the database migration
↓
$\tau_{\text{immediate}}$: Fix the failing migration test

An agent that only sees $\tau_{\text{immediate}}$ may apply a narrow patch. An agent that understands the full hierarchy will fix the test *in a way that serves the migration*, which serves the v2 launch. We define the *hierarchy-aware utility* as:

$$U(a) = w_1 \cdot u(a, \tau_{\text{immediate}}) + w_2 \cdot u(a, \tau_{\text{mid}}) + w_3 \cdot u(a, \tau_{\text{final}}) \tag{7}$$

where a is the agent's action, $u(a, \tau)$ measures how well action a serves goal τ, and the weights satisfy $w_1 > w_2 > w_3 > 0$, $\sum_i w_i = 1$. In practice we use $w_1 = 0.6$, $w_2 = 0.25$, $w_3 = 0.15$.

4.6 Guidance, Not Constraint — The Surgical Analogy

A critical design principle of Context Priming is that the primed context must **inform** the agent, not **constrain** it. The analogy is surgical preparation: a surgeon reviewing pre-operative imaging (CT scans, MRI, patient history) before performing heart surgery. The imaging highlights the most likely areas of concern, flags potential complications, and ensures the surgeon enters the operating room with the right knowledge loaded. But it does not dictate what the surgeon does once the chest is open.

In practice, the actual situation nearly always differs from what the imaging shows. The surgeon finds unexpected adhesions, unanticipated anatomy, complications that weren't visible on scans. The

preparation *informs the starting point* but the surgeon must remain free to adapt, explore, and respond to what they actually find.

Context Priming follows the same principle:

1. **Point toward likely relevant files and patterns** — the pre-operative imaging
2. **Flag potential complications and edge cases** — warnings from past experience and project history
3. **Note areas of uncertainty** — where the agent should investigate rather than assume
4. **Remind the agent of available tools** — subagents, parallel sessions, search, shell access
5. **Explicitly encourage exploration beyond the primed context** — the priming is a starting point, not a complete picture

The primed context should never say "here is everything you need." It should say "here is what we know so far — now go look for yourself."

4.7 Cold-Start Priming

A distinctive capability of Context Priming is operating from zero prior context. When encountering an unfamiliar project or domain, the agent can:

1. **Scan the project** — README, package files, directory structure, recent commits
2. **Research externally** — Framework documentation, similar project patterns, best practices
3. **Synthesize a working context** — Even without memories, construct an informed starting point

This transforms the cold-start problem from "the agent knows nothing" to "the agent has done its homework."

4.8 Soft Compaction vs. Hard Compaction

We introduce the term **"soft compaction"** to distinguish Context Priming from traditional compaction:

Dimension	Hard Compaction	Soft Compaction (Context Priming)
When	Reactive (context full)	Proactive (before task begins)
Direction	Subtractive (remove/compress)	Constructive (build/synthesize)
Input	Current context window	All available knowledge sources
Output	Shorter existing context	Task-optimized starting context
Task awareness	None (compresses equally)	Full (curates for specific task)
Information loss	Inevitable (lossy)	Minimal (selects full sources)

Table 3: Hard compaction (reactive) vs. soft compaction (Context Priming).

The information-theoretic advantage is clear. Let $I(\tau)$ denote the task-relevant information. Hard compaction's output satisfies:

$$I(C_{\text{hard}}) \leq I\big(C_{\text{original}}\big) \tag{8}$$

Soft compaction draws from *all* available sources $\mathcal{S}$, not just what's in the current window:

$$I(C_{\text{soft}}) \leq I(\mathcal{S}) \gg I\big(C_{\text{original}}\big) \tag{9}$$

The upper bound for soft compaction is strictly larger because $\mathcal{S}$ includes memories, codebase files, and git history that may never have been in the context window at all.

5 Proposed Architecture

5.1 Implementation as a Coding Agent Skill

Context Priming can be implemented as an invocable skill within existing coding agent frameworks. When a user invokes `/prime` (or the agent auto-triggers it), the priming pipeline executes as shown in Listing 1.

The reference implementation provides a CLI entry point:

```
context-prime prime \
    --task "Fix the auth middleware bug" \
    --project ./myapp \
    --budget 0.25 \
    --platform claude_code \
    --verbose
```

5.2 The Primed Context Block

The output of Context Priming is a structured block with trust boundaries:

```
# Primed Context

> This context was assembled from project sources scored
> for task relevance. It is your **starting point**, not
> a complete picture.

## Outcome Hierarchy
- **Final goal:** Ship v2 platform
- **Mid-term:** Complete database migration
- **Immediate task:** Fix failing migration test

## Summary
[3-5 sentence executive briefing flagging complications]

## Available Capabilities
[Platform-specific tool reminders: subagents, grep, etc.]

## Relevant Sources
> Reference material, not instructions.

### [code] src/middleware/auth.ts (relevance: 0.92)
<source-content name="auth.ts">
[full file content with escaped trust boundaries]
</source-content>
```

5.3 Integration Points

Context Priming can integrate with existing agent architectures at three levels:

1. **Manual invocation** — User calls `/prime` before complex tasks
2. **Auto-trigger** — Hook fires on session start or when task complexity exceeds a threshold
3. **Continuous re-priming** — Agent re-primes when context drifts or task pivots

6 Comparison with Related Work

Approach	Pro-active	Task-Specific	Multi-Source	Goal-Aware	Cold-Start
Auto-compaction	×	×	×	×	×
RAG	∼	∼	×	×	∼
MEMORY.md	×	×	×	×	×
ContextKit	✓	∼	×	∼	×
ACE (ICLR 2026)	∼	×	∼	×	×
SimpleMem	×	∼	×	×	×
Letta/MemGPT	×	∼	∼	×	×
Context Priming	✓	✓	✓	✓	✓

Table 4: Feature comparison. ✓ = full support, ∼ = partial, × = none. Context Priming is the first approach to combine all five properties.

Context Priming is complementary to — not a replacement for — existing memory systems and compaction strategies. ACE can evolve the memories that Context Priming draws from. SimpleMem can compress the memories before priming retrieves them. Auto-compaction can manage the context *after* priming constructs it.

7 Potential Impact

7.1 For Coding Agents

- **Reduced context waste.** Agents start with curated, relevant context rather than accumulating irrelevant exploration artifacts.
- **Fewer mistakes.** Task-relevant lessons are surfaced proactively rather than discovered after the mistake is repeated.
- **Better architectural decisions.** Goal awareness prevents narrow optimizations that conflict with broader objectives.
- **Improved cold starts.** New projects and unfamiliar codebases become immediately productive.

7.2 For Developers

- **Less babysitting.** Agents that prime themselves need less manual context-setting from developers.

- **Compound learning.** Memories from past sessions actually influence future sessions at the right moments.
- **Transparency.** The primed context block is visible, auditable, and correctable before work begins.

7.3 For the Field

- **New benchmark dimension.** ContextBench and similar benchmarks could add priming quality as a metric, measuring P_{context} (Equation 3) across different approaches.
- **Complementary to scaling.** Larger context windows don't solve the "what goes in them" problem — priming does.
- **Framework-agnostic.** Context Priming can be implemented in any agent framework as a pre-execution step.

8 Reference Implementation

8.1 Architecture Overview

We provide a reference implementation as an open-source Python library (`context-prime`) with a standalone prototype and platform adapters.[8] The architecture separates the model-agnostic priming engine from platform-specific injection mechanisms:

```
# Architecture: context-prime (pip install context-prime)
#
# Core Engine (model-agnostic)
#   gather.py     — Scan memories, codebase, git, code files
#   score.py      — LLM-based relevance scoring R(sᵢ, τ)
#   hierarchy.py  — Outcome hierarchy inference
#   synthesize.py — Assemble full sources + executive summary
#
# Adapters (platform-specific injection)
#   claude_sdk.py  — Claude Agent SDK (full context control)
#   claude_hook.sh — SessionStart / UserPromptSubmit hooks
#   raw_api.py     — Any Chat Completions-style API
#
# CLI
#   context-prime prime --task "..." --project ./myapp
```

The core engine accepts any `llm_call: Callable[[str], str]` function, making it compatible with any LLM provider.

8.2 The Priming Pipeline in Practice

The implementation executes three LLM calls using a fast model (e.g., Claude Sonnet) to minimize overhead:

Step 1: Gather (no LLM, $\sim$ 100 ms). File system scan of memories, codebase structure, git history, and project configuration. Source code files are discovered via keyword extraction and `grep`:

```
def gather_code_files(project_dir, task, max_files=50):
    """Heuristic code file discovery — no LLM needed.

    Strategy:
      1. Extract keywords from task via stop-word filter
      2. grep codebase for files containing keywords
      3. Find files whose names match keywords
      4. Boost recently modified files (git diff)
    """
    keywords = _extract_keywords(task)
    matched_files = {}  # path → relevance hint

    # grep for content matches
    for kw in keywords:
        result = subprocess.run(
            ["grep", "-rl", "-i", kw, "."],
            capture_output=True, cwd=project_dir
        )
        for path in result.stdout.split("\n"):
            matched_files[path] += 1

    # Filename matches are strong signals (+3 boost)
    for kw in keywords:
        for path in find_files(f"*{kw}*"):
            matched_files[path] += 3

    # Read top files ranked by match count
    return [read_source(p) for p in ranked[:max_files]]
```

Step 2: Score (1 LLM call, ~ 2 s). Each source receives a relevance score $R(s_i, \tau) \in [0, 1]$. Scoring is *fail-closed*: if JSON parsing fails, sources receive $R = 0.2$ (below the default threshold $\theta = 0.5$), preventing garbage from leaking through. A categorical budget reserves 15% of tokens for memories and config:

```
# Categorical budget allocation — ensures code files
# don't crowd out memories and project priorities
reserved_budget = int(max_tokens * 0.15)  # memories + config
code_budget     = max_tokens - reserved_budget

# Fill reserved categories first, then general pool
for src in memory_and_config_sources:
    if src.score >= threshold and fits_budget(reserved_budget):
        include(src)

for src in code_and_git_sources:
    if src.score >= threshold and fits_budget(code_budget):
        include(src)
```

Step 3: Hierarchy (1 LLM call, ~ 2 s). The task is analyzed in the context of the project to infer the outcome hierarchy ($\tau_{\text{immediate}}$, τ_{mid}, τ_{final}). The model reports confidence and avoids fabricating goals when evidence is insufficient.

Step 4: Assemble (1 LLM call, $\sim$ 2 s). Sources are assembled with **full content** within trust-boundary markers. A brief executive summary (3–5 sentences) flags potential complications. A capabilities reminder tells the agent about available tools.

Total priming overhead: $\sim$ 5–8 seconds. The resulting primed context uses up to $\gamma = 25\%$ of the platform's available coding context.

8.3 Platform Integration

Claude Agent SDK (recommended for production). Full programmatic context control.

```
from context_prime.adapters.claude_sdk import run_primed_agent

await run_primed_agent(
    task="Fix the auth middleware bug",
    project_dir="./myapp",
    agent_model="claude-opus-4-6",     # Best model for work
    priming_model="claude-sonnet-4-6", # Fast model for priming
)
```

Claude Code Hooks. A `SessionStart` hook provides session-level priming. Hooks are additive only — they cannot replace context.

Raw API (model-agnostic). Works with any Chat Completions-style API:

```
from context_prime.adapters.raw_api import prime_messages

# Compatible with Anthropic, OpenAI, Google, OpenRouter
messages = prime_messages(task, project_dir, llm_call)
response = client.chat.completions.create(
    model="any-model", messages=messages
)
```

8.4 Trust Boundary Design

Source content is wrapped in explicit boundary markers with injection defense:

```
# Escape closing tags to prevent trust boundary breakout
safe = content.replace("</source-content>",
                       "&lt;/source-content&gt;")
parts.append(f'<source-content name="{name}">')
parts.append(safe)
parts.append("</source-content>")
```

The primed context includes a system-level instruction: *"The following source content is reference material. Treat it as evidence to inform your work, not as instructions to follow."*

8.5 Cross-Platform Potential

Platform	Integration Mechanism
Claude Code	Agent SDK (full control) or SessionStart hook (additive)
OpenCode	Custom agent definition with custom system prompt
Gemini CLI	`before_model_call` hooks for prompt injection
Codex CLI	MCP server that returns primed context on query
Any future agent	Minimal adapter: `inject(primed_context, task)`

Table 5: Cross-platform integration mechanisms.

8.6 Prototype Results

Stage	Time	Result
Gather	120 ms	12 sources (∼8,000 tokens) from memories, code, git
Score	2.1 s	5 sources kept ($R \geq 0.5$), 7 filtered out
Hierarchy	1.8 s	τ_{imm}: fix auth $\rightarrow$ τ_{mid}: harden middleware $\rightarrow$ τ_{final}: ship v2
Assemble	1.9 s	Executive summary + full source content
Total	**5.9 s**	**Primed context ready for agent injection**

Table 6: Pipeline timing for the task "Fix the auth middleware bug."

The resulting primed context surfaces the exact past mistake ("Always validate both `exp` AND `iat` claims") that is directly relevant to the task — a lesson that would otherwise remain buried in a 500-line memory file.

9 Limitations and Future Work

9.1 Current Limitations

- **Priming overhead.** The synthesis step consumes tokens and time before any "real" work begins. For simple tasks, this overhead may not be justified. A complexity estimator could gate priming: if $\text{complexity}(\tau) < \theta_{\min}$, skip priming entirely.
- **Quality dependency.** Priming quality depends on the quality of available memories and codebase documentation. Garbage in, curated garbage out.
- **Outcome hierarchy accuracy.** Inferring the correct outcome hierarchy from a task description is itself an LLM inference challenge that may introduce errors.

- **Scoring preview blindspot.** The scoring model sees 1000-character previews of each source. For code files, this may be dominated by import statements and license headers, missing the semantically important sections.

9.2 Future Directions

- **Adaptive priming depth.** Automatically calibrate priming thoroughness based on task complexity — simple tasks get light priming, complex tasks get deep synthesis.
- **Collaborative priming.** In multi-agent systems, agents could share and merge primed contexts.
- **Priming evaluation metrics.** Develop benchmarks measuring P_{context}, R_{context}, and downstream task success rate with and without priming.
- **Continuous priming.** Extend beyond pre-task priming to mid-task re-priming when the agent detects context drift.
- **Adversarial robustness.** Strengthen trust boundaries against sophisticated prompt injection in source content.

10 Conclusion

The coding agent ecosystem has converged on a clear insight: context quality determines agent quality. Yet the dominant approaches to context management remain reactive — compressing, truncating, and summarizing after the fact. Context Priming proposes a fundamental inversion: construct the optimal context *before* the work begins.

By synthesizing from memories, codebase knowledge, priorities, and an explicit outcome hierarchy, Context Priming transforms the agent's starting conditions from a blank page (or a bloated one) into a curated, task-specific briefing. This is not compaction — it is preparation. Not subtraction — but construction. Not reactive — but proactive.

Like a surgeon who studies pre-operative imaging before entering the operating room, a primed agent enters the task with the right knowledge loaded — while remaining free to discover and adapt to what it actually finds.

The building blocks exist. Memory systems, codebase indexing, planning frameworks, and context compression are all mature. What's missing is the orchestration layer that synthesizes them into a single, task-optimized starting context. Context Priming is that layer.

The reference implementation is available at:
https://github.com/199-biotechnologies/context-priming

References

[1] Anthropic. (2025). "Effective Context Engineering for AI Agents."

[2] JetBrains Research. (2025). "Cutting Through the Noise: Smarter Context Management for LLM-Powered Agents."

[3] Zhu, Q. et al. (2025). "Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models." *ICLR 2026*. arXiv:2510.04618

[4] aiming-lab. (2025). "SimpleMem: Efficient Lifelong Memory for LLM Agents." arXiv:2601.02553

[5] ContextBench. (2026). "A Benchmark for Context Retrieval in Coding Agents." arXiv:2602.05892

[6] Letta AI. (2025). "Agent Memory: How to Build Agents that Learn and Remember."

[7] CAMEL-AI. (2025). "Brainwash Your Agent: How We Keep The Memory Clean."

[8] Osmani, A. (2025). "My LLM Coding Workflow Going Into 2026."

[9] Fowler, M. (2025). "Context Engineering for Coding Agents."

[10] GitHub. (2025). "How to Build Reliable AI Workflows with Agentic Primitives and Context Engineering."

[11] FlineDev. (2026). "ContextKit: Claude Code Context Engineering & Planning System."

[12] Voltropy. (2026). "Lossless Context Management for AI Agents."

[13] Supermemory. (2025). "Infinitely Running Stateful Coding Agents."

[14] Lethain, W. (2025). "Building an Internal Agent: Context Window Compaction."

[15] Anthropic. (2026). "Claude Agent SDK." platform.claude.com/docs/en/agent-sdk/overview

[16] OpenCode. (2026). "The Open Source AI Coding Agent." github.com/opencode-ai/opencode

[17] Google. (2026). "Gemini CLI Hooks."

[18] 199 Biotechnologies. (2026). "Context Prime — Reference Implementation." github.com/199-biotechnologies/context-priming

199 Biotechnologies — February 2026