

Context Engineering

Master the art of managing what goes into a language model's context window — from tokens and system prompts to tool calls and memory strategies.

Based on the outline from latentpatterns.com

March 2026

Table of contents

1	Context Engineering	4
1.1	Who This Book Is For	4
1.2	Why “Context Engineering”?	4
1.3	What You’ll Learn	5
1.4	Chapters	5
1.5	The Central Insight	6
1.6	How to Read This Book	7
1.7	Credits	7
1.8	References and Further Reading	7
I	Foundations	9
2	Chapter 1: Tokens and Inference	10
2.1	What Are Tokens?	10
2.2	Inference Is Stateless	11
2.3	Key Takeaways	13
2.4	References	13
3	Chapter 2: The Real Size of Your Context Window	14
3.1	Marketing Numbers vs. Engineering Reality	14
3.2	Why Models Fail at Length	15
3.3	The Smart Zone and the Dumb Zone	16
3.4	Measuring Your Context Budget	16
3.5	Key Takeaways	17
3.6	References	17
II	The Messages Array	19
4	Chapter 3: Anatomy of the Messages Array	20
4.1	Slot 0 — The System Prompt	20
4.2	Slot 1 — The Harness Prompt	21
4.3	Slot 2 — Project Context Files	21
4.4	Slot 3 — MCP Servers and Agent Skills	22
4.5	Slot 4 — Your Prompt	24
4.6	Key Takeaways	25

4.7	References	25
5	Chapter 4: Dynamic Allocation — Tool Calling	26
5.1	Tool Calls as Memory Allocations	26
5.2	A Real Agent Session	28
5.3	Key Takeaways	29
5.4	References	29
III	Scaling Patterns	30
6	Chapter 5: Sub-Agents — Managed Runtimes for AI	31
6.1	Sub-Agents Are Not About Personas	31
6.2	The Test Runner Problem	32
6.3	Designing Sub-Agent Boundaries	33
6.4	Key Takeaways	35
6.5	References	35
7	Chapter 6: Message Passing — The Erlang OTP of AI	36
7.1	Context Windows Are Actors	36
7.2	Designing the Message Protocol	37
7.3	Supervision and Failure	39
7.4	Key Takeaways	40
7.5	References	40
IV	Architecture	42
8	Chapter 7: The Ralph Wiggum Loop	43
8.1	What Is the Ralph Wiggum Loop?	43
8.2	The Three-Phase Funnel	44
8.3	Specs and Plans — The Persistent Memory	45
8.4	Backpressure and Guardrails	46
8.5	Greenfield vs. Brownfield	47
8.6	Failure Modes and Criticisms	48
8.7	Key Takeaways	49
8.8	References	49
9	Chapter 8: Context Management Strategies	50
9.1	The <code>malloc</code> Without <code>free</code>	50
9.2	Why Compaction Is Dangerous	51
9.3	Better Strategies	52
9.4	Key Takeaways	55
9.5	References	55
10	Summary	56
11	Publication History	57
11.1	Credits	57

11.2 Timeline 57

11.3 Design Decisions 57

11.4 License 58

Chapter 1

Context Engineering

Master the art of managing what goes into a language model’s context window — from tokens and system prompts to tool calls and memory strategies.

Source: latentpatterns.com/courses/context-engineering

1.1 Who This Book Is For

Software developers, ML engineers, and architects who build applications on top of LLMs — especially agent-based systems. You should be comfortable with API calls to language models and have experience (even brief) with tools like Claude Code, Cursor, Copilot, or LangChain.

1.2 Why “Context Engineering”?

The term was popularized in June 2025 by Tobi Lutke (Shopify CEO) — “the art of providing all the context for the task to be plausibly solvable by the LLM” — and Andrej Karpathy, who elaborated: “context engineering is the delicate art and science of filling the context window with just the right information for the next step.” It supersedes “prompt engineering” because in production systems, the prompt is the least of your concerns. The real work is managing everything else that goes into context: tool definitions, retrieved documents, conversation history, project conventions, and compaction strategies.

1.3 What You’ll Learn

The context window is the single most important resource in any LLM application. It determines what the model knows, how well it reasons, and when it fails. Most developers treat it as a black box. This book treats it as a system to be engineered.

By the end, you’ll be able to:

- Calculate the real (not advertised) context budget for any model
 - Design system prompts, project context, and tool sets for maximum signal-per-token
 - Architect agent systems that stay in the “smart zone” across long sessions
 - Use sub-agents, message passing, and fresh-context patterns to scale beyond single-window limits
 - Apply memory management principles from systems programming to context management
-

1.4 Chapters

1.4.1 Part I: Foundations

Chapter	Topic	Sections
1. Tokens and Inference	How LLMs consume text and why inference is stateless	What Are Tokens? Inference Is Stateless
2. The Real Size of Your Context Window	Why advertised context lengths are marketing, not engineering	Marketing vs. Reality Why Models Fail at Length The Smart Zone and the Dumb Zone Measuring Your Context Budget

1.4.2 Part II: The Messages Array

Chapter	Topic	Sections
3. Anatomy of the Messages Array	The slot-by-slot structure of what goes into context	System Prompt Harness Prompt Project Context Files MCP Servers and Tools Your Prompt
4. Dynamic Allocation — Tool Calling	How tool calls grow context and what it costs	Tool Calls as Memory Allocations A Real Agent Session

1.4.3 Part III: Scaling Patterns

Chapter	Topic	Sections
5. Sub-Agents — Managed Runtimes for AI	Context isolation through disposable child windows	Not About Personas The Test Runner Problem Designing Boundaries
6. Message Passing — The Erlang OTP of AI	Context windows as actors with message-passing semantics	Context Windows Are Actors Designing the Message Protocol Supervision and Failure

1.4.4 Part IV: Architecture

Chapter	Topic	Sections
7. The Ralph Wiggum Loop	Crash-only agent design — fresh-context iteration for autonomous coding	What Is It? The Three-Phase Funnel Specs and Plans Backpressure Greenfield vs. Brownfield Failure Modes
8. Context Management Strategies	Memory management for LLMs — allocation, compaction, and beyond	The malloc Without free Why Compaction Is Dangerous Better Strategies

1.5 The Central Insight

Treat the context window as you would treat memory in a systems programming language: **a finite, precious resource that requires deliberate management.**

The techniques in this book — from token budgeting to sub-agent isolation to supervision trees — are the context engineering equivalent of the memory management discipline that separates reliable systems from fragile ones.

1.6 How to Read This Book

- **Sequential:** Chapters build on each other. Chapter 1’s token economics inform Chapter 2’s budget calculations, which inform Chapter 3’s slot analysis, and so on.
 - **Reference:** Each chapter is self-contained enough to revisit independently. Cross-references link to prerequisite concepts.
 - **Practical:** Every concept is tied to concrete numbers, real examples, and actionable patterns. No theory without application.
 - **PDF:** The full book is available as a [PDF download](#) for offline reading.
-

1.7 Credits

Written by [Claude](#) (Anthropic), directed and edited by a human author. Outline from [Latent Patterns](#). See [publication history](#) for details.

Licensed under [CC BY-ND 4.0](#).

1.8 References and Further Reading

1.8.1 Papers

- Hsieh, C.-P., et al. (2024). “RULER: What’s the Real Context Size of Your Long-Context Language Models?” [arXiv:2404.06654](#)
- Liu, N. F., et al. (2023). “Lost in the Middle: How Language Models Use Long Contexts.” [arXiv:2307.03172](#)
- Lewis, P., et al. (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” [arXiv:2005.11401](#)
- Packer, C., et al. (2023). “MemGPT: Towards LLMs as Operating Systems.” [arXiv:2310.08560](#)
- Paulsen, T. (2025). “Context Is What You Need: The Maximum Effective Context Window.” [arXiv:2509.21361](#)
- Chroma Research. (2025). “Context Rot.” research.trychroma.com/context-rot

1.8.2 Books

- Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.
- Kernighan, B. & Ritchie, D. (1988). *The C Programming Language*, 2nd Ed. Prentice Hall.

- Brooks, F. (1975). *The Mythical Man-Month*. Addison-Wesley.

1.8.3 Articles and Blog Posts

- Anthropic. (2025). [“Effective Context Engineering for AI Agents.”](#)
- Anthropic. (2025). [“How We Built Our Multi-Agent Research System.”](#)
- Huntley, G. (2025). [“Ralph Wiggum as a ‘Software Engineer.’”](#)
- Farr, C. (2025). [“The Ralph Wiggum Playbook.”](#)
- Manus. (2025). [“Context Engineering for AI Agents.”](#)
- Cognition. (2025). [“Don’t Build Multi-Agents.”](#)
- Schmid, P. (2025). [“Single vs Multi-Agent System?”](#)
- Martin, L. (2025). [“Context Engineering for Agents.”](#)

1.8.4 Tools and Documentation

- [Anthropic — Claude API Documentation](#)
- [Anthropic — Model Context Protocol \(MCP\)](#)
- [Anthropic — Claude Code](#)
- [OpenAI — Tokenizer](#)
- [OpenAI — tiktoken](#)
- [Google — SentencePiece](#)

Part I

Foundations

Chapter 2

Chapter 1: Tokens and Inference

2.1 What Are Tokens?

2.1.1 Why tokens, not characters

LLMs don't process raw text. Before anything happens, text is split into **tokens** — subword units that balance vocabulary size with sequence length. Tokens are the atomic unit of LLM computation. The model reads tokens. The model generates tokens. Pricing is per token. Context limits are in tokens.

Everything in context engineering is measured in tokens.

Common words like “the” or “hello” become single tokens. Rare words like “defenestration” get split into pieces (“def”, “en”, “est”, “ration” or similar). Different models use different tokenization schemes and vocabulary sizes, so the same text can produce different token counts depending on the model.

2.1.2 Token IDs are just integers

Under the hood, a language model is a function that takes an array of integers and produces the next integer. That's it. The text you type is converted to token IDs before the model sees it, and the token IDs the model outputs are converted back to text before you see them. The sentence "Hello, world!" becomes something like [9906, 11, 1917, 0].

This means everything in the context window — your prompt, the system instructions, the conversation history, the tool call results — becomes a single flat stream of integer tokens. The model doesn't “see” the structure you see. It sees numbers.

2.1.3 The 4:1 rule of thumb

In English prose, **1 token 4 characters 0.75 words**. That means 1,000 tokens cover roughly 750 words, and a 100K-token context holds approximately 75,000 words — about the length of a short novel.

This ratio varies wildly by content type, though. **Code** is token-dense because whitespace, punctuation, and rare identifiers (variable names, imports) all fragment into more tokens per character. **Non-English languages** often require more tokens per word, since BPE training corpora are English-heavy — CJK characters, Arabic, and many others tokenize less efficiently. **JSON and XML** are extremely token-expensive: every brace, bracket, quote, colon, and key name eats tokens, and structural overhead can consume 3-5x more tokens than the equivalent data in plain text.

2.1.4 Why this matters for context engineering

Every character you put into context has a token cost. A verbose system prompt wastes tokens. Bloated tool outputs waste tokens. Wrapping data in JSON when plain text would suffice wastes tokens.

Understanding token economics is the foundation of context engineering. You cannot manage what you cannot measure.

2.1.5 Practical examples

Consider a few concrete cases. The string "Hello, world!" typically becomes 4 tokens (Hello, ,, world, !). A typical JSON API response might use 3-5x more tokens than the same data formatted as plain text or markdown. The string {"name": "Alice", "age": 30} costs more tokens than Alice, age 30 — and both convey the same information to the model.

Tools for counting tokens: - [OpenAI Tokenizer](#) — interactive web tool - [tiktoken](#) — Python library for programmatic token counting - Anthropic's API returns token counts in every response's `usage` field

Get in the habit of measuring. Don't guess token counts — compute them.

2.2 Inference Is Stateless

2.2.1 The fundamental truth

LLM inference is stateless. The server retains nothing between API calls. Every request must contain the complete conversation state.

There is no session. There is no memory. There is only the messages array.

2.2.2 The messages array IS the state

Every API call sends the full conversation:

```
{
  "model": "...",
  "system": "You are a helpful assistant.",
  "messages": [
    {"role": "user", "content": "What is 2+2?"},
    {"role": "assistant", "content": "4."},
    {"role": "user", "content": "And if you add 3?"}
  ]
}
```

(API formats vary — Anthropic uses a top-level `system` field, OpenAI uses `role: "system"` in the messages array. The concept is the same: a system instruction plus a sequence of user/assistant turns.)

The model sees this entire array on every call. It doesn't "remember" saying "4." — it re-reads it. The fourth message only makes sense because the third message is present. Remove any message and the model loses that context entirely.

2.2.3 The illusion of memory

Chat UIs — ChatGPT, Claude.ai, Gemini — maintain the messages array client-side and replay it on every request. This creates the illusion of a continuous conversation.

But the implication is critical: **every previous message costs tokens on every subsequent call**. A 50-message conversation replays all 50 messages when you send message 51. The conversation gets more expensive with every exchange — linearly in tokens, quadratically in compute (due to attention).

2.2.4 Implications for context engineering

These mechanics lead to four consequences that define the discipline.

First, **context is a finite, depletable resource** — not free storage. Every token you add to context is a token you can't use for something else. Second, **every token stays forever** unless you actively manage it. Old messages, stale tool results, and irrelevant context don't expire — they accumulate. Third, **the cost of context is paid on every inference call**. A 10K-token system prompt is charged on every single API call in the conversation. Fourth, and most importantly, **this**

is why context engineering exists: to manage this scarce resource intelligently. What goes in? What stays? What gets summarized? What gets dropped? These are the core questions.

2.2.5 Current context window sizes (early 2026)

Model	Context Window	Max Output
GPT-5 (OpenAI)	400K tokens	128K
Claude Opus 4.6 (Anthropic)	200K (1M beta)	64K
Claude Sonnet 4.6 (Anthropic)	200K (1M beta)	64K
Gemini 2.5 Pro (Google)	1M tokens	64K
Llama 4 Scout (Meta)	10M tokens	—
DeepSeek R1	128K tokens	64K

These numbers look generous. But as we'll see in [Chapter 2](#), they are marketing numbers — not engineering reality. Effective context is always smaller than advertised context.

2.3 Key Takeaways

- **Tokens are the currency of context.** Everything is measured in tokens.
- **1 token 4 English characters.** But format matters enormously — JSON costs 3-5x more than plain text for the same data.
- **Inference is stateless.** The messages array is replayed in full on every call.
- **Context is a finite resource** that gets more expensive with every exchange.

2.4 References

- OpenAI Tokenizer: <https://platform.openai.com/tokenizer>
- OpenAI tiktoken library: <https://github.com/openai/tiktoken>
- Anthropic API Documentation: <https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching>

Chapter 3

Chapter 2: The Real Size of Your Context Window

3.1 Marketing Numbers vs. Engineering Reality

Model providers advertise context windows of 128K, 200K, 1M, even 10M tokens. These numbers represent the theoretical maximum input the model can accept. They do not represent the context size at which the model performs well. The gap between advertised and effective context is substantial, and understanding it is essential.

The most common benchmark for testing long context is **needle-in-a-haystack (NIAH)**: hide a specific fact in a long document, then ask the model to retrieve it. Models score near-perfect on these tests at their full advertised context lengths. But NIAH is misleading — it only tests single-fact retrieval from a mostly irrelevant context. Real tasks require reasoning over multiple pieces of information scattered throughout context.

The **RULER benchmark** (Hsieh et al., 2024) provides a much more rigorous test across four categories:

1. **Single retrieval (NIAH)**: Find one needle — models do well
2. **Multi-key retrieval**: Find multiple needles — performance drops significantly
3. **Tracing**: Follow chains of reasoning through context — steep degradation
4. **Aggregation**: Synthesize information across the full context — worst performance

The key finding from RULER is striking: of 17 models tested claiming 32K+ context, only 4 maintained acceptable performance at their claimed length across all task categories. GPT-4 (the best performer) still dropped 15.4 percentage points from 4K to 128K context. Mixtral (32K claimed) collapsed from 94.9% at 4K to 44.5% at 128K.

The practical takeaway is clear. A 200K-token model might perform reliably only up to ~65-100K tokens for complex tasks. A 128K model might degrade noticeably past 50K. The advertised number is the ceiling, not the floor.

3.2 Why Models Fail at Length

Transformer attention is theoretically over all tokens, but in practice, models develop biases. They attend more strongly to tokens at the beginning (primacy) and end (recency) of context. This **attention degradation** is the root cause of long-context failures.

The landmark “**Lost in the Middle**” paper (Liu et al., 2023) demonstrated this concretely. Information placed in the middle of long contexts is significantly less likely to be retrieved or used correctly. Models form a “U-shaped” attention curve — strong at start and end, weak in the middle. In a 20-document QA task, accuracy was ~75% when the answer was at position 1, dropped to ~55% at position 10 (middle), and recovered to ~72% at position 20. That is a 20+ percentage-point gap from a positional shift alone.

The RULER benchmark identified four specific failure modes that explain how this degradation manifests:

1. **Distraction failure:** Irrelevant context actively interferes with retrieval
2. **Multi-hop failure:** Can’t chain together multiple pieces of information
3. **Pattern tracking failure:** Loses track of repeated or sequential patterns
4. **Aggregation failure:** Can’t synthesize or count across many distributed items

These failure modes matter especially for agent workloads, which are exactly the kind of task that triggers them. An agent session accumulates tool results, code snippets, error messages, and instructions — all interleaved. The critical instruction from 50 messages ago is exactly the kind of “middle context” that gets lost.

In practice, these academic failure modes show up as recognizable **practitioner-level failure patterns**. **Context poisoning** occurs when the model hallucinates a function name early on; that hallucination stays in context and becomes “ground truth,” and the model references its own mistake in later turns, compounding the error. **Instruction attenuation** happens as context fills with tool results and code — the system prompt’s instructions lose “weight,” and the model stops following formatting rules, forgets safety constraints, or abandons the original goal. **Context confusion** arises when multiple versions of the same file enter context (the original read, then a modified version), and the model blends them into a plausible-but-wrong hybrid.

These are not hypothetical. They are the everyday experience of developers using coding agents.

3.3 The Smart Zone and the Dumb Zone

The core concept here is to divide the context window into two regions. **The Smart Zone** is the portion of the context window where the model performs reliably — typically the first ~40-50% of the advertised context length. **The Dumb Zone** is the remaining capacity where performance degrades progressively. The model still accepts tokens, but reasoning quality, instruction following, and retrieval accuracy decline.

This leads to a practical guideline: **the 40% rule of thumb**. For complex, multi-step tasks (like agent workloads involving multi-hop reasoning), aim to keep total context utilization under ~40% of the advertised window. This is a conservative design target, not an empirical hard line — simpler tasks (single retrieval, straightforward generation) can work fine at higher utilization. But agent workloads hit the worst case. Applied to current models:

- 200K window → target 80K of actual context
- 128K window → target 50K of actual context
- 1M window → target 400K of actual context

Research from Chroma Research (2025) confirms this degradation is real and universal. They tested 18 frontier models and found that every model degrades as input length increases — there are no exceptions. Their data shows approximately 2% effectiveness loss per 100K tokens added for Claude models (which decayed slowest among those tested). This degradation is continuous, not a cliff — there’s no safe length below which context is “free.”

Recognizing when your agent has crossed into the dumb zone is a practical skill. The signs include: repeating actions it already took, forgetting instructions from earlier in the conversation, making errors on tasks it handled correctly earlier, “hallucinating” about what happened in the session, and tool calls becoming less targeted and more exploratory.

When you detect dumb-zone behavior, the answer is not “give it more context.” The answer is to reset: start a fresh context window with only the essential state. This is the foundation of patterns we’ll cover in [Chapter 5](#), [Chapter 6](#), and [Chapter 7](#).

3.4 Measuring Your Context Budget

Your **context budget** is your total available tokens minus all fixed allocations. This is what’s actually available for the dynamic parts of your application — the conversation, tool results, and reasoning.

Several categories of **fixed allocations** consume tokens on every request. System prompts typically run 500-2,000 tokens. Harness and framework prompts add 1,000-5,000 tokens. Project context

(AGENTS.md and similar files) contributes 500-3,000 tokens. Tool definitions cost 200-500 tokens per tool, multiplied by the number of tools — if you have 20 MCP tools, that’s 4,000-10,000 tokens just for tool schemas.

The budget formula brings this together:

$$\text{Effective budget} = (\text{Advertised window} \times 0.4) - \text{Fixed allocations}$$

Here’s a worked example with Claude Sonnet 4.6 (200K window):

Allocation	Tokens
Smart zone ($200\text{K} \times 0.4$)	80,000
System prompt	−1,500
Harness prompt	−3,000
CLAUDE.md	−2,000
Tool definitions (15 tools)	−5,000
Effective budget	68,500

That’s your real budget for conversation + tool results + reasoning. It sounds like a lot — until a single file read returns 5,000 tokens, a test run dumps 10,000 tokens, and you’re 10 exchanges deep. Budget awareness must be designed into your agent, not bolted on after. This is why sub-agents ([Chapter 5](#)) and fresh-context patterns ([Chapter 7](#)) exist.

3.5 Key Takeaways

- Advertised context lengths are marketing. Effective context is 40-50% of the number on the box.
- NIAH tests are misleading. Use RULER-style benchmarks to understand real model capabilities.
- Models lose information in the middle of context (“Lost in the Middle” effect).
- Calculate your real context budget: $(\text{window} \times 0.4)$ minus fixed allocations.
- Design for the smart zone. When you exceed it, reset — don’t push through.

3.6 References

- Hsieh, C.-P., et al. (2024). “RULER: What’s the Real Context Size of Your Long-Context Language Models?” arXiv:2404.06654. <https://arxiv.org/abs/2404.06654>
- Liu, N. F., et al. (2023). “Lost in the Middle: How Language Models Use Long Contexts.” arXiv:2307.03172. <https://arxiv.org/abs/2307.03172>

- Kamradt, G. (2023). “Needle In A Haystack - Pressure Testing LLMs.” https://github.com/gkamradt/LLMTest_NeedleInAHaystack
- Li, Y., et al. (2024). “Long-context LLMs Struggle with Long In-context Learning.” [arXiv:2404.02060](https://arxiv.org/abs/2404.02060).
- Chroma Research. (2025). “Context Rot: How Increasing Input Tokens Impacts LLM Performance.” <https://research.trychroma.com/context-rot>
- Paulsen, T. (2025). “Context Is What You Need: The Maximum Effective Context Window.” [arXiv:2509.21361](https://arxiv.org/abs/2509.21361).

Part II

The Messages Array

Chapter 4

Chapter 3: Anatomy of the Messages Array

This chapter walks through the messages array slot by slot, from index 0 to the end. Think of it as a memory layout — each slot has a purpose and a cost.

4.1 Slot 0 — The System Prompt

The **system prompt** is the model’s standing instructions — who it is, how it should behave, what constraints apply. In OpenAI’s API, this is a message with `role: "system"` (or `role: "developer"` in newer models). In Anthropic’s Claude API, it’s a top-level `system` parameter separate from the messages array. Either way, it occupies the same conceptual slot: the first thing the model sees.

It matters because the system prompt occupies the most privileged position in context. Due to primacy effects (attention is strongest at the start), the system prompt has outsized influence on model behavior compared to the same text placed later.

An effective system prompt has a consistent anatomy: identity and role (1-2 sentences), core behavioral constraints (what to do and not do), output format specifications, and domain-specific knowledge or rules. Keep it tight. Every token here is paid on every request.

That cost adds up quickly. A 2,000-token system prompt on a model charging \$3/M input tokens, across 1,000 API calls = \$6 just for the system prompt. In agent workloads with hundreds of calls per session, system prompt bloat is a real cost.

Watch out for common anti-patterns: repeating the same instruction 3 different ways “for emphasis” (wastes tokens, doesn’t help), including examples that could be retrieved on-demand instead, and

stuffing knowledge that should be in a retrieval system.

4.2 Slot 1 — The Harness Prompt

When you use an agent framework or tool (Claude Code, Cursor, Copilot, LangChain, etc.), the framework injects its own instructions into the messages array. This is the **harness prompt**. You typically don't see it, but it's consuming your context.

A typical harness prompt contains tool usage instructions (“To read a file, use the Read tool...”), safety constraints and behavioral rules, output formatting requirements, and environment descriptions (OS, working directory, available tools). It can range from 1,000 to 8,000+ tokens depending on the framework.

There's a clear trade-off here. Off-the-shelf harnesses (Claude Code, Cursor) provide robust, well-tested harness prompts but you can't control their token cost. Custom agent implementations let you optimize the harness but require more engineering.

The harness prompt is a “hidden tax” on your context budget. If you're building a custom agent and your harness is 5,000 tokens, that's 5,000 tokens you can't use for actual work on every single call. Audit it. Measure it. Trim it.

Most frameworks don't expose the harness prompt directly, so inspecting it requires some effort. You can ask the model “What are your system instructions?” (unreliable), intercept API calls with a proxy, read the framework's source code (for open-source tools), or check token usage — if your prompt is 100 tokens but the API reports 3,000 input tokens, the harness is the difference.

4.3 Slot 2 — Project Context Files

Project context files are project-specific instructions that are automatically discovered and injected into context. Every major tool has its own variant — `CLAUDE.md` (Claude Code), `AGENTS.md` (Codex CLI, an open standard), `.cursorrules` (Cursor), `.github/copilot-instructions.md` (Copilot). They tell the model about your specific project — conventions, architecture, key patterns.

These files are discovered through a hierarchy. In Claude Code, for example:

1. `~/.claude/CLAUDE.md` — user-level defaults
2. `./CLAUDE.md` — project root
3. `./src/CLAUDE.md` — subdirectory overrides

These get concatenated and injected into the messages array.

4.3.1 Signal-per-token optimization

This is the most important concept for project context files. Every token must carry maximum information. Compare:

Bad (47 tokens):

```
When working on this project, please make sure to always use TypeScript
instead of JavaScript. We prefer TypeScript because it provides better
type safety and developer experience.
```

Good (9 tokens):

```
Use TypeScript. Never plain JavaScript.
```

Same instruction. 5x fewer tokens. The model understands both equally well.

Knowing what to include — and what to leave out — is critical. Good candidates are: tech stack and key dependencies (with versions that matter), file/folder structure conventions, testing patterns and commands, common gotchas specific to this codebase, and architecture decisions the model needs to respect.

Conversely, you should exclude information the model can discover by reading files, generic best practices the model already knows, long examples (use references to files instead), and anything that rarely applies (retrieve on-demand instead).

A 2026 study from ETH Zurich evaluating AGENTS.md files found a surprising result: LLM-generated context files actually *decreased* task success rates by ~3% and increased costs by 20%+. Human-written files improved success by ~4% — but only when limited to information the agent genuinely can't infer from the codebase. The takeaway: context files that merely restate discoverable information (architecture overviews, directory structures) add cost without benefit. Include only non-discoverable, operationally significant information: custom build commands, project-specific gotchas, unconventional patterns.

4.4 Slot 3 — MCP Servers and Agent Skills

MCP (Model Context Protocol) is a standard protocol for exposing tools (functions the model can call) and resources (data the model can read) to LLMs. Tool definitions are JSON schemas that describe available functions, their parameters, and expected behavior.

4.4.1 The constant-allocation problem

Tool definitions consume tokens on EVERY request, whether or not the tools are used. This is because the model needs to “know” what tools are available in order to decide when to call them.

Typical tool definition sizes: - Simple tool (1-2 params): ~100-200 tokens - Medium tool (3-5 params): ~200-400 tokens - Complex tool (nested params, enums): ~400-800 tokens

10 tools $\times$ 300 tokens avg = 3,000 tokens of fixed overhead. 50 tools $\times$ 300 tokens avg = 15,000 tokens — that’s a substantial chunk of your smart zone.

A real-world example from MCP token cost analysis drives this home: a developer setup with GitHub (35 tools), Slack (11 tools), Sentry (5 tools), and a few more MCP servers — 58 tools total — consumed ~55,000 tokens. That’s ~28% of a 200K context window burned on tool schemas alone, before any conversation begins.

4.4.2 Eager vs. lazy loading

There are three strategies for managing tool definitions. **Eager loading** (the default in most frameworks) includes all tool definitions in every request. It’s simple, but wasteful if you have many tools and most are rarely used. **Lazy loading** starts with a minimal tool set and provides a “discover tools” meta-tool that can load additional tool definitions on demand. This saves tokens but adds latency (extra round trip to discover tools before using them). The **hybrid** approach eagerly loads core tools (file read/write, search, execute) while lazy-loading specialized tools (database, deployment, specific APIs).

Claude Code implements deferred tool loading concretely via a `ToolSearch` meta-tool. Tools marked with `defer_loading: true` are excluded from the initial prompt entirely — Claude sees only the `ToolSearch` tool plus core tools. When Claude needs a deferred tool, it searches by keyword; the API returns 3-5 matching tool definitions that auto-expand into full schemas. This auto-enables when MCP tools would consume >10% of context. The results are significant: 85-95% reduction in startup token overhead, and it doesn’t break prompt caching (deferred tools aren’t in the prefix). At 50+ eagerly-loaded tools, model accuracy for tool selection drops to ~49%; `ToolSearch` recovers it to ~74%.

When designing tools for token efficiency, keep tool descriptions concise and use clear, self-documenting parameter names instead of long descriptions. Avoid overly granular tools — fewer, more capable tools beat many tiny ones. And always consider: does this tool need to be available on every call, or just sometimes?

4.4.3 A note on prompt caching

Fixed allocations (system prompt, harness, tools, project context) are paid on every request — but they don’t have to cost full price. Both Anthropic and OpenAI offer **prompt caching**: if the beginning of your input matches a previous request, the cached portion is processed at a steep discount (Anthropic charges cached tokens at ~10% of uncached rate: \$0.30/MTok vs \$3/MTok for Sonnet).

This doesn’t change the context engineering picture — cached tokens still consume context window space. But it dramatically reduces the dollar cost of fixed allocations. The practical implication: optimize your fixed context for signal-per-token (context window efficiency) but also for cache hit rate (cost efficiency). Keep the prefix of your messages array stable across calls — any change invalidates the cache from that point forward.

Manus’s production experience reinforces this: use append-only context with deterministic serialization. Even a single-token change in the prefix invalidates everything after it. Their practices include keeping system prompt + tools + project context in a stable prefix, adding conversation turns only at the end, and explicitly marking cache breakpoints. With Claude Sonnet, the difference is 10x: \$0.30/MTok cached vs \$3/MTok uncached.

4.5 Slot 4 — Your Prompt

After system prompt, harness, project context, and tool definitions are allocated, the remaining space is your actual context budget — for the user’s prompt, conversation history, and tool results.

Revisiting the budget calculation from Chapter 2, here’s what a heavier tool setup looks like:

Allocation	Tokens
Smart zone ($200\text{K} \times 0.4$)	80,000
System prompt	−1,500
Harness prompt	−4,000
Project context	−1,500
Tool definitions (20 tools)	−6,000
Available for conversation	67,000

With 67,000 tokens of real budget, your initial user prompt matters. A 500-token prompt leaves 66,500 for conversation and tool results. A 5,000-token prompt (pasting a large file inline) leaves 62,000. If your first prompt is 20,000 tokens (a whole codebase dump), you’ve used 30% of your budget before the first response.

This leads to the golden rule of context-aware prompt design: don't put in context what can be retrieved. Instead of pasting a file into your prompt, ask the agent to read it with a tool. The file still enters context, but only when needed, and you have the option to manage that allocation later.

4.6 Key Takeaways

- The messages array is a memory layout with fixed and dynamic regions.
- Slots 0-3 (system prompt, harness, project context, tools) are fixed costs paid on every call.
- Signal-per-token is the design principle for all fixed-cost context.
- Tool definitions are surprisingly expensive — audit and optimize your tool set.
- Your actual budget for conversation is what's left after fixed allocations.

4.7 References

- Anthropic. “Model Context Protocol (MCP) Specification.” <https://modelcontextprotocol.io/>
- Anthropic. “Claude Code Documentation — CLAUDE.md.” <https://docs.anthropic.com/en/docs/claude-code>
- Anthropic. “Prompt Caching.” <https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching>
- Anthropic. “Effective Context Engineering for AI Agents.” <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
- OpenAI. “Function Calling Guide.” <https://platform.openai.com/docs/guides/function-calling>
- Cursor. “cursorsrules Documentation.” <https://docs.cursor.com>
- GitHub Copilot. “Customizing Copilot Instructions.” <https://docs.github.com/en/copilot/customizing-copilot>
- MMNTM. “The MCP Tax: Hidden Costs of Model Context Protocol.” <https://www.mmntm.net/articles/mcp-context-tax>

Chapter 5

Chapter 4: Dynamic Allocation — Tool Calling

5.1 Tool Calls as Memory Allocations

If the messages array is memory, then every tool call is a `malloc()`. Each tool interaction adds at minimum three entries to the array:

1. The assistant's message containing the tool call (the function name + arguments)
2. The tool result (returned by your application)
3. The assistant's response interpreting the result

The token cost of a single tool call cycle breaks down like this:

Assistant decides to call a tool:

```
"I'll read the file src/auth.ts"  
→ ~50-100 tokens (reasoning + tool_use block)
```

Tool call payload:

```
{"name": "read_file", "input": {"path": "src/auth.ts"}}  
→ ~30-50 tokens
```

Tool result (the file contents):

```
→ 200-5,000+ tokens (depends entirely on what comes back)
```

Assistant processes the result:

```
"The auth module exports three functions..."
```

→ 100-500 tokens

Total for one tool call: 400-6,000+ tokens. A single file read can consume as many tokens as the entire system prompt.

To see what this looks like in the actual API payload, here's the messages array after one tool call cycle:

```
[
  {"role": "system", "content": "You are a coding assistant..."},
  {"role": "user", "content": "What's in config.json?"},
  {"role": "assistant", "content": [
    {"type": "tool_use", "id": "call_1", "name": "read_file",
     "input": {"path": "config.json"}}
  ]},
  {"role": "user", "content": [
    {"type": "tool_result", "tool_use_id": "call_1",
     "content": "{\n  \"db_host\": \"localhost\",\n  \"db_port\": 5432,\n  \"auth\": {\n\"provid"
  ]},
  {"role": "assistant", "content": "The config has database settings (localhost:5432) and OAuth"}
]
```

Five messages from one question. And every one of them stays in the array for every subsequent API call. The JSON config content — which the model already interpreted — is re-sent and re-processed on every future turn.

This creates an **accumulation problem**: tool results are append-only. Once a 3,000-token file is read into context, it stays there for the rest of the session. Read 10 files? That's potentially 30,000 tokens of file content that persists in your context forever (within that session).

To visualize how the array grows during a typical interaction:

[system]	→ 1,500 tokens	(cumulative: 1,500)
[harness]	→ 3,000 tokens	(cumulative: 4,500)
[user: "Fix the login bug"]	→ 20 tokens	(cumulative: 4,520)
[assistant: "Let me look..."]	→ 100 tokens	(cumulative: 4,620)
[tool_call: search_code]	→ 80 tokens	(cumulative: 4,700)
[tool_result: 15 matches]	→ 2,000 tokens	(cumulative: 6,700)
[assistant: "Found it, reading..."]	→ 150 tokens	(cumulative: 6,850)
[tool_call: read_file]	→ 50 tokens	(cumulative: 6,900)
[tool_result: file contents]	→ 4,000 tokens	(cumulative: 10,900)

[assistant: "I see the bug..."]	→ 300 tokens	(cumulative: 11,200)
[tool_call: edit_file]	→ 200 tokens	(cumulative: 11,400)
[tool_result: "success"]	→ 30 tokens	(cumulative: 11,430)
[assistant: "Fixed. Let me test"]	→ 100 tokens	(cumulative: 11,530)
[tool_call: run_tests]	→ 50 tokens	(cumulative: 11,580)
[tool_result: test output]	→ 8,000 tokens	(cumulative: 19,580)
[assistant: "All tests pass."]	→ 80 tokens	(cumulative: 19,660)

In just one bug fix cycle: ~15,000 tokens of dynamic context consumed. That's ~22% of a 68K effective budget (from Chapter 2's calculation) — gone in one task.

5.2 A Real Agent Session

Consider what happens when an agent handles a more complex request — say, “Add user authentication to this Express app.” A realistic multi-task session unfolds in four phases.

Phase 1: Understanding covers exploring the codebase. The agent searches for existing auth patterns (~2,000 tokens), reads package.json (~800 tokens), reads app.ts (~3,000 tokens), and reads existing middleware (~2,500 tokens). Running total: ~8,300 new tokens.

Phase 2: Implementation is where code gets written. Creating auth middleware costs ~500 tokens (tool call + result), creating the user model ~400 tokens, updating routes ~600 tokens, and creating the login endpoint ~500 tokens. Running total: ~10,300 new tokens.

Phase 3: Testing is where context pressure really builds. Running tests produces ~5,000 tokens of output. Reading error details adds ~2,000 tokens. A fix costs ~300 tokens, but then re-running tests dumps another ~5,000 tokens. Another fix at ~300 tokens, another test run at ~5,000 tokens. Running total: ~28,200 new tokens.

Phase 4: Cleanup rounds out the session with the model now deep in context. Updating the README adds ~500 tokens, and a final test run produces another ~5,000 tokens. Running total: ~33,700 new tokens.

The **total context at end** is ~11,500 (fixed allocations from Chapter 2) + 33,700 (dynamic) = **~45,200 tokens**. Still in the smart zone for a 200K model, but only for a single feature addition. Two or three more tasks in the same session, and you're in the dumb zone.

Notice where context pressure hits hardest: **test output**. Testing alone consumed ~20,000 tokens — more than half the session's dynamic context. Each test run dumps its full output into the array. This is the #1 context pressure point in agent sessions. ([Chapter 5](#) introduces the solution: sub-agents for test running.)

5.2.1 Signs of session degradation

When a session starts running long, you’ll see predictable symptoms:

1. The agent re-reads files it already read
2. It forgets constraints you stated earlier
3. Tool calls become less precise (searching broadly instead of targeted reads)
4. It starts apologizing and “trying again” without changing approach
5. Code quality drops — more bugs, less coherent architecture

When you notice these signs, the most productive action is often to start a new session, not to “remind” the agent. Reminding adds more tokens to an already-stressed context. Starting fresh (with a well-designed spec) gives the model a clean smart zone to work in. This is the bridge to [Chapter 7](#).

5.3 Key Takeaways

- Every tool call is a memory allocation — it grows the context permanently within a session.
- A single file read can cost 3,000-5,000 tokens. Test output is the #1 context hog.
- A typical agent session for one feature can consume 30,000-50,000 tokens of dynamic context.
- Context pressure shows up as degraded behavior: repetition, forgotten instructions, imprecise tool use.
- The solution isn’t “more context” — it’s fresh context with the right state carried forward.

5.4 References

- Anthropic. “Tool Use (Function Calling) Documentation.” <https://docs.anthropic.com/en/docs/build-with-claude/tool-use>
- OpenAI. “Function Calling Guide.” <https://platform.openai.com/docs/guides/function-calling>
- Schick, T., et al. (2023). “Toolformer: Language Models Can Teach Themselves to Use Tools.” arXiv:2302.04761.
- Patil, S., et al. (2023). “Gorilla: Large Language Model Connected with Massive APIs.” arXiv:2305.15334.

Part III

Scaling Patterns

Chapter 6

Chapter 5: Sub-Agents — Managed Runtimes for AI

6.1 Sub-Agents Are Not About Personas

When people hear “multi-agent,” they typically think of role-playing — “one agent is the architect, another is the coder, another is the reviewer.” This is the wrong mental model. Personas are a prompt engineering trick. Sub-agents solve a systems engineering problem.

What sub-agents actually are: a sub-agent is a separate context window that runs independently, does work, and returns a result to the parent. The parent’s context is protected from the sub-agent’s internal work. Think of it as a function call in programming:

Parent context: `"Run the tests and tell me if they pass."`

Sub-agent context (separate window):

- `Receives: "Run the tests"`
- `Executes: runs test suite`
- `Sees: 50,000 tokens of test output`
- `Returns: "All 47 tests pass. 3 were skipped (reason: ...)."`

Parent context receives: the 50-token summary, NOT the 50,000-token output

The analogy that best captures this is managed runtimes or containers. Each sub-agent gets its own isolated memory (context window), runs its workload, and communicates only through well-defined interfaces (the prompt in, the result out). The parent is like the orchestrator — it delegates work and collects results.

This matters enormously for context engineering. Without sub-agents, every operation’s full output enters the parent’s context and stays there. With sub-agents, the parent only pays for the summary. This is the single most effective technique for extending how much work an agent can do in a session.

6.2 The Test Runner Problem

Consider the problem in concrete terms: running a test suite from an agent session.

Without sub-agent:

Parent context before test: 25,000 tokens

Test output added to context: +15,000 tokens

Parent context after test: 40,000 tokens

Tokens consumed for the test information: 15,000

Tokens the parent actually needs: ~200 (pass/fail + summary)

Waste: 14,800 tokens (98.7%)

With sub-agent:

Parent context before test: 25,000 tokens

Sub-agent runs tests internally (its own 15,000 tokens - separate window)

Sub-agent returns summary: +200 tokens

Parent context after test: 25,200 tokens

Tokens consumed in parent: 200

Savings: 14,800 tokens

Over 3 test runs in a session, that’s 44,400 tokens saved. That’s the difference between staying in the smart zone and entering the dumb zone.

This follows a **“schedule-a-future” pattern** — think of sub-agent delegation as scheduling a future/promise:

1. Parent creates a task description (the “promise”)
2. Sub-agent resolves the promise (does the work)
3. Parent receives the resolved value (the summary)

The parent doesn’t need to know HOW the work was done — just the result.

Test running is just the canonical example. Other high-token operations benefit from sub-agent delegation in the same way. **Large file analysis** — “Read these 5 files and tell me which ones

handle authentication” — lets the sub-agent read all files while the parent gets just a list. **Code search** — “Find all usages of the deprecated API” — lets the sub-agent search while the parent gets locations. **Documentation generation** — “Write docs for this module” — lets the sub-agent read code and write docs while the parent gets confirmation. **Dependency analysis** — “What are the security vulnerabilities in our dependencies?” — lets the sub-agent run the audit while the parent gets a summary.

6.3 Designing Sub-Agent Boundaries

The key question for delegation is: “Will this operation produce output I don’t need verbatim in the parent context?” If yes, delegate to a sub-agent. If no — for example, a small file edit where you need to see the exact result — run it directly.

6.3.1 The Delegation Decision Matrix

Operation	Expected tokens	Need verbatim?	Decision
Read a 50-line config file	~500	Yes (need to edit it)	Direct
Run test suite	5,000-50,000	No (need pass/fail)	Sub-agent
Search codebase for pattern	2,000-10,000	No (need locations)	Sub-agent
Edit a specific function	~300	Yes	Direct
Analyze 10 files for patterns	10,000-50,000	No (need findings)	Sub-agent
Read a single error log	~200	Maybe	Direct (small enough)

6.3.2 Designing the Message Interface

The **inbound message** (parent to sub-agent) should be specific about what you need back. “Run the tests and return: (1) pass/fail count, (2) names of any failing tests, (3) the error message for each failure.” Don’t just say “run the tests.”

The **outbound message** (sub-agent to parent) should be results-focused, not process-focused. The parent doesn’t need “First I read the file, then I noticed...” — it needs “3 tests failed: test_auth (missing token), test_login (timeout), test_signup (duplicate email).”

6.3.3 The Overhead Trade-Off

Sub-agents aren't free. Each sub-agent pays the full fixed allocation cost (system prompt + harness + tools). There's latency overhead from spinning up a new context and making additional API calls. The parent needs tokens to describe the task and process the result.

Rule of thumb: sub-agents are worth it when the delegated operation would add 2,000+ tokens to the parent context that the parent doesn't need verbatim.

There is also a **token explosion risk** to consider. Sub-agents save tokens in the *parent* context, but they still consume tokens globally — and parallelism multiplies cost fast. Practitioners report hitting Claude Pro plan limits in as little as 15 minutes when running 5 parallel sub-agents. Anthropic documents that multi-agent workflows use 4-7x more tokens than single-agent approaches; full Agent Teams architectures can reach ~15x. Information flows only parent-to-child, never child-to-child — if two sub-agents need to coordinate, the parent must relay messages, adding further token cost on both sides.

On **nesting depth**: sub-agents can spawn their own sub-agents, but keep it shallow. Each level adds latency and fixed overhead. In practice, 2 levels (parent, child, grandchild) is usually the maximum useful depth.

6.3.4 Read vs. Write: The Key Heuristic

The most useful heuristic for when multi-agent works comes from Phil Schmid: “The important distinction isn't single vs multi-agent... it is whether your task primarily involves reading or writing.”

Read tasks (research, analysis, code review, search) parallelize well — sub-agents gather information independently and results combine naturally. Multi-agent shines here. **Write tasks** (code generation, document authoring, system design) create coordination nightmares — parallel writers make conflicting assumptions that an integrator can't reconcile. Single agent is usually better.

The data backs this up: Google/MIT research found that parallelized write tasks degraded performance by 39-70%, while parallelized read tasks improved performance by 80.9%.

6.3.5 The Anti-Pattern Warning

Cognition, the team behind Devin, published “Don't Build Multi-Agents” — arguing that parallel sub-agents making independent decisions leads to conflicting implicit choices. Their recommendation: sub-agents should only handle **well-defined questions** (like “run these tests”), never substantive decision-making. Keep decision-making centralized in the parent. Use sub-agents for information gathering and execution, not for planning or architectural choices.

Cognition’s **Flappy Bird example** makes this concrete: they asked parallel sub-agents to build the game. One sub-agent built a Mario-style background. Another built a bird that was visually and mechanically inconsistent with that background. The integrator agent couldn’t reconcile the conflicting outputs. Root cause: “conflicting assumptions not prescribed upfront.” The sub-agents each made reasonable but incompatible creative decisions — exactly the kind of implicit choice that can’t be parallelized safely.

6.4 Key Takeaways

- Sub-agents are about context isolation, not personas.
- The test runner problem is the canonical example: 15,000 tokens of test output becomes 200 tokens of summary.
- Delegate when the operation produces tokens the parent doesn’t need verbatim.
- **Read tasks parallelize well; write tasks don’t.** Use sub-agents for research, analysis, and search. Keep code generation and design decisions in the parent.
- Design explicit message contracts: be specific about what you need back.
- Sub-agents have real overhead (4-7x tokens for multi-agent) — use them when the token savings justify the cost.

6.5 References

- Anthropic. “Claude Code — Sub-agents Documentation.” <https://code.claude.com/docs/en/sub-agents>
- Cognition. “Don’t Build Multi-Agents.” <https://cognition.ai/blog/dont-build-multi-agents>
- Manus. “Context Engineering for AI Agents: Lessons from Building Manus.” <https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus>
- Wu, Q., et al. (2023). “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.” arXiv:2308.08155.
- Hong, S., et al. (2023). “MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework.” arXiv:2308.00352.
- Schmid, P. (2025). “Single vs Multi-Agent System?” <https://www.philschmid.de/single-vs-multi-agents>
- Google Research. (2025). “Towards a Science of Scaling Agent Systems.” <https://research.google/blog/towards-a-science-of-scaling-agent-systems-when-and-why-agent-systems-work/>
- Anthropic. “How We Built Our Multi-Agent Research System.” <https://www.anthropic.com/engineering/multi-agent-research-system>

Chapter 7

Chapter 6: Message Passing — The Erlang OTP of AI

7.1 Context Windows Are Actors

The **Erlang OTP actor model** is built on isolated processes that share nothing and communicate exclusively through message passing. Processes can spawn children, forming hierarchies. Failure is expected — processes crash, and supervisors restart them. This maps directly to AI agent systems.

7.1.1 The Mapping

Erlang/OTP	AI Agents
Process	Context window
Process mailbox	Messages array / prompt
Message passing	Task description → result summary
No shared memory	Each context window is isolated
Spawn a child process	Launch a sub-agent
Process state	Accumulated context within a session
Process crash	Context window exceeds smart zone / errors out
Supervisor	Parent agent that manages sub-agents

This analogy matters because Erlang solved the same fundamental problem 40 years ago — how do you build reliable systems from unreliable components that can't share state? The answer: isolate them, let them communicate through messages, and design for failure.

The “no shared memory” constraint is real. Two context windows cannot share state. There is no global variable, no shared database they both see in real-time. The only way to move information between context windows is to explicitly send it — just like Erlang processes. This constraint, while it feels limiting, is actually what makes the system reliable.

Data transfer between agents follows **copy semantics**. When you send data to a sub-agent, you’re copying it. The sub-agent gets its own version. If the sub-agent modifies its understanding, the parent doesn’t know unless the sub-agent explicitly sends back a message. This is exactly how Erlang works: messages are copied, never referenced.

7.2 Designing the Message Protocol

The **inbound message** (parent to sub-agent) is the sub-agent’s prompt — the task description. Design it like an API contract.

Bad message (vague, will produce unpredictable results):

Look at the code and fix any issues.

Good message (specific contract):

Task: Run the test suite for the auth module.

Command: `npm test -- --testPathPattern=auth`

Return format:

- Total tests: <number>
- Passed: <number>
- Failed: <number>
- For each failure: test name, expected vs actual, file:line
- Do NOT include passing test details

The **outbound message** (sub-agent to parent) is the return value. Design for minimum tokens, maximum signal.

Bad return (process-focused, verbose):

First I ran `npm test` and saw that there were some failures.

I looked at the output and found that three tests failed.

The first failure was in `test_login` where...

[500 tokens of narration]

Good return (results-focused, structured):

Total: 47 | Passed: 44 | Failed: 3

Failures:

1. `test_login_expired_token` - Expected: 401, Got: 500 (`src/auth.test.ts:42`)
2. `test_refresh_missing_header` - Expected: 400, Got: undefined (`src/auth.test.ts:78`)
3. `test_signup_duplicate` - Expected: 409, Got: 200 (`src/auth.test.ts:103`)

7.2.1 Protocol Design Principles

Four principles govern good message protocol design:

1. **Specify the return format in the inbound message:** Don't hope the sub-agent returns what you need. Tell it exactly what format to use.
2. **Results, not narration:** The parent needs data, not a story about how the sub-agent got the data.
3. **Failure information over success details:** If everything passed, a one-liner is sufficient. Detail the failures.
4. **Structured over prose:** Structured formats (lists, tables, key-value pairs) are easier for the parent model to parse and reason about.

7.2.2 The `todo.md` Attention Manipulation Pattern

Manus discovered a powerful technique for maintaining coherence during long agent runs (~50+ tool calls). Their agents create and continuously update a `todo.md` file throughout execution — checking off completed items and adding new ones as the task evolves. This pushes current objectives into the model's most recent attention span, directly combating the lost-in-the-middle problem. The filesystem serves as what Manus calls “unlimited, persistent, and directly operable” extended context. It's a message the agent sends to its future self, using the filesystem as the medium.

7.2.3 The “Agent as Tool” Mental Model

Phil Schmid offers an alternative framing: rather than thinking of sub-agents as members of an org chart (“the architect agent talks to the coder agent”), treat them as deterministic function calls: `call_planner(goal="...")`. Harness frameworks like Hugging Face's `smolagents` spin up a temporary sub-agent loop, let it run, and return structured output that's immediately usable by the caller — no different from calling any other tool. This mental model keeps the protocol clean: input in, output out, no ongoing relationship or state to manage.

The sub-agent's system prompt can formalize the protocol directly:

```
You are a test runner agent. Execute the requested tests and return
results in this exact format:
```

SUMMARY: <total> tests, <passed> passed, <failed> failed

FAILURES:

- <test_name>: <expected> vs <actual> (<file>:<line>)

NOTES: <any relevant observations, max 2 sentences>

7.3 Supervision and Failure

Erlang’s “**let it crash**” philosophy says: don’t write defensive code to prevent every possible failure inside a process. Instead, let processes crash and have supervisors that detect the crash and restart the process. This produces simpler, more reliable code than trying to handle every edge case.

Applied to AI agents, sub-agents can fail in several ways: they exceed their context window, they hallucinate and produce wrong results, they get stuck in loops, the API call errors out (rate limit, timeout, etc.), or they misunderstand the task.

7.3.1 Supervision Strategies

Borrowing from Erlang OTP, there are three core supervision strategies.

one_for_one: If a sub-agent fails, restart just that sub-agent. This is the simplest and most common strategy.

Parent delegates: "Run auth tests"

Sub-agent fails (timeout)

Parent response: Spawn new sub-agent, same task

one_for_all: If one sub-agent fails, restart all sub-agents. Use this when sub-agents’ work is interdependent.

Parent delegates: "Build frontend" + "Build backend" (in parallel)

Frontend sub-agent fails

Parent response: Restart both (backend may depend on frontend decisions)

rest_for_one: If a sub-agent fails, restart it and all sub-agents that were started after it. Use this when there’s a sequential dependency.

7.3.2 Practical Supervision in AI Agents

The simplest supervisor pattern is **retry with the same prompt**. This often works because LLM outputs are non-deterministic — the same prompt may succeed on retry. If the sub-agent failed because of ambiguity, **retry with a modified prompt** to clarify the task. If retries fail, **escalate**

to the parent, which can try a different approach entirely. In all cases, set a **retry budget** — typically 2-3 attempts maximum — to avoid infinite loops.

7.3.3 Supervision Trees

Just as Erlang builds hierarchies of supervisors, complex agent systems can build hierarchies:

```
Orchestrator (top-level)
|-- Feature Agent (supervisor)
|   |-- Code Writer (worker)
|   |-- Test Runner (worker)
|   \-- Reviewer (worker)
|-- Feature Agent (supervisor)
|   |-- Code Writer (worker)
|   \-- Test Runner (worker)
\-- Integration Test Agent (worker)
```

Each supervisor manages its workers. If a worker fails, the supervisor handles it. If the supervisor can't handle it, it escalates to the orchestrator.

The most important property of this architecture is **failure isolation**: a sub-agent's failure doesn't corrupt the parent's context. If a sub-agent hallucinates wildly, those hallucinations exist only in the sub-agent's context window. The parent sees only the result message (or the failure signal).

7.4 Key Takeaways

- Context windows map directly to Erlang actors: isolated processes communicating through messages.
- Design explicit message protocols — specify input format AND output format.
- Results-focused returns beat narration. Structure beats prose.
- “Let it crash” applies: design for sub-agent failure with retry and supervision strategies.
- Failure isolation is the key benefit — a sub-agent's problems don't infect the parent's context.

7.5 References

- Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD Thesis. (The definitive Erlang/OTP reference.)
- Armstrong, J. (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf.
- Erlang/OTP Design Principles: https://www.erlang.org/doc/system/design_principles

- Hewitt, C., Bishop, P., & Steiger, R. (1973). “A Universal Modular ACTOR Formalism for Artificial Intelligence.” IJCAI.
- Agha, G. (1986). *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.
- Manus. “Context Engineering for AI Agents.” <https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus>
- Schmid, P. (2025). “Context Engineering Part 2.” <https://www.philschmid.de/context-engineering-part-2>

Part IV

Architecture

Chapter 8

Chapter 7: The Ralph Wiggum Loop

8.1 What Is the Ralph Wiggum Loop?

The **Ralph Wiggum Loop** was created by Geoffrey Huntley in early 2025 and named after Ralph Wiggum from The Simpsons — a character who lives entirely in the present moment with no memory of what came before. In its purest form, it’s a bash loop:

```
while ;; do cat PROMPT.md | claude-code ; done
```

Each new context window is Ralph: capable and completely unaware of previous iterations. The pattern went viral in late 2025. Huntley’s philosophical framing: “My institutional knowledge is in the specifications file.” Code is clay on a pottery wheel. The spec is the real asset.

The core idea is straightforward: instead of fighting context degradation, embrace the reset. Run your agent in a loop of fresh context windows. Each iteration starts clean, reads the spec, does one task, saves progress, and exits. The next iteration picks up where it left off — not from memory, but from files.

This works because each iteration gets the full smart zone. There’s no accumulated cruft from tool calls, no degraded attention. The model is at peak performance every time. Huntley puts it bluntly: “Compaction is the devil” — instead of lossy summarization, you get lossless persistence via the filesystem.

The key insight is that the specification file IS the memory. The context window is disposable. This inverts the usual mental model where people try to preserve the conversation. Instead: preserve the plan, discard the conversation.

This is, at heart, a **crash-only design** — a concept from systems programming introduced by Candea and Fox (2003). In crash-only software, you don’t write elaborate defensive code to keep

a process running forever. Instead, you design the program to crash safely, read its state from durable storage, and restart fresh. The Ralph Loop applies the same principle to AI sessions: the context window is designed to be disposable. It crashes (resets), reads its state from the spec files on disk, and starts fresh with full capability. There is no attempt to gracefully degrade or salvage a deteriorating context.

This connects to the “let it crash” philosophy discussed in Chapter 6 (Message Passing), drawn from Erlang’s supervision trees — but applied at the session level rather than the sub-agent level. An Erlang supervisor restarts a failed child process; the Ralph Loop restarts the entire conversation. Both reject the premise that uptime requires defensive complexity. The alternative — trying to keep a single context window alive through compaction, summarization, and eviction — is the traditional “defensive” approach, and Chapter 8 covers those strategies. But Huntley’s bet is that crashing clean beats degrading gracefully.

8.2 The Three-Phase Funnel

Huntley structures work as three distinct phases, each with its own prompt and loop:

8.2.1 Phase 1: Requirements (conversational, not looped)

Before any loop runs, spend 30+ minutes in a dedicated planning conversation using **bidirectional prompting**:

1. Describe what you want to build
2. Ask the model to ask YOU questions (surface hidden assumptions)
3. Answer the questions
4. Ask the model to generate more questions based on your answers
5. Repeat until the model has no more questions
6. Have the model write the specification

This produces specs dramatically more complete than what you’d write alone. The model surfaces edge cases, architectural decisions, and requirements you hadn’t considered. The output is one or more spec files in a `specs/` directory — one topic per file, passing the “single-sentence without conjunctions” test.

8.2.2 Phase 2: Planning (looped, but no code)

Run the planning loop: `./loop.sh plan` with `PROMPT_plan.md`

The agent performs gap analysis — comparing specs against existing code. It outputs a prioritized `IMPLEMENTATION_PLAN.md`. **No code is written. No commits.** This is pure analysis.

For existing codebases, this phase is where **spec convergence** happens: the agent studies the codebase, maps what exists, and the plan accounts for existing patterns and constraints. Without this phase, the agent will duplicate existing functionality or violate established conventions.

8.2.3 Phase 3: Building (looped, one task per iteration)

Run the build loop: `./loop.sh` with `PROMPT_build.md`

Each iteration: agent reads the spec + plan, picks the most important unchecked task, implements it, runs tests, commits, and exits. The loop restarts fresh.

“One thing per loop” — Huntley’s most emphatic principle. Each iteration does exactly one task. Trust the agent to pick which one. You control what’s in the plan; the agent controls execution order.

8.2.4 The File Structure

```
project-root/
|-- loop.sh                # Bash orchestrator (mode selection, iteration limits)
|-- PROMPT_plan.md        # Planning mode instructions
|-- PROMPT_build.md       # Building mode instructions
|-- AGENTS.md             # Operational guide (~60 lines max)
|-- IMPLEMENTATION_PLAN.md # Living state - checkboxes, progress, notes
|-- specs/                # One requirement file per topic
|   |-- authentication.md
|   \-- data-model.md
\-- src/
```

8.3 Specs and Plans — The Persistent Memory

Since each iteration starts fresh, the specification and plan files must contain everything the agent needs to pick up where the last iteration left off. These files are the only thing that persists between iterations — design them accordingly.

Checkbox-based progress tracking is the mechanism that ties iterations together. Structure the implementation plan as a checklist:

Implementation Plan

- [x] Set up project structure
- [x] Create database schema

- [x] Implement user model
- [] Add authentication middleware ← current
- [] Create login/register endpoints
- [] Add JWT token refresh
- [] Write integration tests
- [] Update API documentation

Notes

- JWT library: using jose (iteration #4 discovered jsonwebtoken has ESM issues)
- Auth middleware must check both cookie and Authorization header (spec update from iteration 4)

Good specs follow several design principles. They must be **self-contained** — the spec alone must be enough to continue work, with no dependency on “remembering” a previous conversation. They should be **concise**, around ~5,000 tokens per spec file, since every spec loads into every iteration’s context and bloat wastes your smart zone. Each file should cover **one topic** — if you need “and” to describe it, split it. Completion criteria should be **machine-verifiable**: “If you can’t define ‘done’ in terms a test suite can verify, Ralph can’t stop.” Finally, specs should be **append-friendly**, with a “Notes” section where iterations log discoveries for future iterations.

Plans should be treated as disposable. “A plan that drifts is cheaper to regenerate than to salvage.” When reality diverges from the plan, switch back to planning mode and regenerate. Don’t try to manually fix a stale plan.

8.4 Backpressure and Guardrails

The most underrated aspect of the Ralph Loop isn’t the loop itself — it’s the environment it runs in.

“Backpressure beats direction”: instead of telling the agent what to do, engineer an environment where wrong outputs get rejected automatically. Tests, linters, type-checkers, and build validation create hard gates. If the agent produces bad code, the backpressure system catches it — the agent sees the failure on its next iteration and corrects course.

Start with **hard gates** — tests and builds are deterministic, so begin there. A passing test suite is a stronger signal than any prompt instruction. “Your codebase is stronger evidence than your instructions” — existing code patterns steer the agent more than prompts do.

Git serves as the safety net. Each iteration commits independently, so `git reset --hard` is an acceptable recovery strategy. The version history is your undo button.

You should also **cap iterations** — set explicit limits (5-50 depending on scope) to prevent runaway costs and overbaking. Don't let the loop run indefinitely.

Always **run in sandboxes** — Docker containers or isolated environments. Huntley: “It's not if it gets popped, it's when.”

For heavier workloads, **use sub-agents** — up to many parallel sub-agents for reads, but only 1 for build/tests. Serializing validation prevents backpressure collapse — if multiple agents run tests simultaneously, they can't tell whose changes broke what.

8.5 Greenfield vs. Brownfield

Huntley famously said “There's no way in heck would I use Ralph in an existing code base.” This is directionally right — **naive** Ralph on existing code creates duplicates, ignores existing patterns, and causes regressions.

But it's not absolute. Huntley's own formalized methodology (his GitHub repo) instructs the agent to “compare specifications against existing source code to identify gaps” and warns “don't assume not implemented.” The planning phase is explicitly brownfield-aware.

There are documented successes on existing code. Ronin Consulting ported a 900K-line game engine from Windows to macOS in 4 days. A developer who'd never seen the code used HumanLayer's ACE-FCA on a 300K LOC Rust codebase (BAML) to fix a bug in 1 hour and add 35K LOC of features in 7 hours. Dan Malone migrated a 3-year-old Next.js/React framework across major versions — 271 commits in 3 days.

What makes brownfield work is that the research/planning phase must be proportionally larger and the iteration scope proportionally smaller:

1. **Research first:** Agent explores the existing codebase before any spec is written
2. **Spec only the change area:** Don't reverse-engineer the whole system. Spec coverage grows incrementally where changes happen
3. **Capture conventions in AGENTS.md:** Build commands, architecture patterns, testing conventions — information the agent can't infer from code alone
4. **Existing test suites as guardrails:** Brownfield has an advantage here — tests already exist to catch regressions

This connects to the read-vs-write heuristic from Chapter 5 (Sub-Agents): brownfield work involves heavy writes into an existing codebase, which resist parallelization. Greenfield projects can fan out reads and even writes across sub-agents because there's no shared mutable state to conflict with. Brownfield, by contrast, demands serial iteration — exactly what the Ralph Loop provides.

Without this approach, the agent “ignored the notes that these were descriptions of existing classes, it just took them as a new specification and generated them all over again, creating duplicates” (Thoughtworks evaluation of spec-kit).

8.6 Failure Modes and Criticisms

The Ralph Loop is not a silver bullet. Know its failure modes:

Overbaking happens when the loop runs too long without supervision, producing unrequested features. Huntley documented “bizarre emergent behavior” — post-quantum cryptography support appearing in a project that didn’t need it.

Oscillation is the back-and-forth trap: “It fixed a type error in `utils.ts` by breaking an import in `main.ts`. Then it fixed `main.ts` by reverting the change in `utils.ts`.” No progress is made. Iteration caps and human review break the cycle.

Duplicate implementations occur without “don’t assume not implemented” guardrails — the agent rebuilds existing functionality. This is critical in brownfield work.

Plan staleness sets in when the plan drifts from reality but isn’t regenerated. The solution: treat plans as disposable — switch to planning mode when drift is detected.

There is also a **task complexity ceiling**. METR’s 2025 research found agent success rates drop from near-100% on sub-4-minute tasks to less than 10% on tasks requiring 4+ hours of human effort. Ralph doesn’t change this ceiling — it helps you stay under it by decomposing large tasks.

Cost is a real factor. 50 iterations on a medium codebase cost \$50-100+ in API credits. Each iteration re-reads the spec and relevant files. Budget for this.

Finally, **code structure quality** deserves attention. GitClear’s 2025 analysis found AI-assisted development correlates with increased copy-paste code and decreased refactoring. Ralph-generated code runs but may lack structural coherence. Huntley targets “90% done” — expect human refinement.

8.6.1 Prompt language that matters

Practitioners discovered specific phrasings that trigger better agent behavior:

- “**study**” (not “read” or “look at”) — triggers deeper analysis
- “**don’t assume not implemented**” — prevents duplicate functionality
- “**only 1 subagent for build/tests**” — serializes validation
- “**if functionality is missing then it’s your job to add it**” — prevents learned helplessness

- “capture the why” — produces better documentation

8.7 Key Takeaways

- The Ralph Wiggum Loop is a three-phase funnel: Requirements → Planning (no code) → Building (one task per loop).
- The specification is the asset. Code is clay. The context window is disposable.
- Backpressure (tests, linting, type-checking) steers the agent more than prompt instructions.
- It works on existing codebases — but the research phase must be proportionally larger and iteration scope smaller.
- Know the failure modes: overbaking, oscillation, duplication, plan staleness, complexity ceiling.
- Budget: ~5K tokens per spec, AGENTS.md under 60 lines, cap iterations at 5-50, expect \$50-100 per 50 iterations.

8.8 References

- Candea, G. & Fox, A. (2003). “Crash-Only Software.” 9th Workshop on Hot Topics in Operating Systems (HotOS IX).
- Huntley, G. (2025). “Ralph Wiggum as a ‘Software Engineer.’” <https://ghuntley.com/ralph/>
- Huntley, G. (2025). “Everything is a Ralph Loop.” <https://ghuntley.com/loop/>
- Huntley, G. (2025). “How to Ralph Wiggum.” <https://github.com/ghuntley/how-to-ralph-wiggum>
- Farr, C. (2025). “The Ralph Wiggum Playbook.” <https://paddo.dev/blog/ralph-wiggum-playbook/>
- HumanLayer. (2025). “A Brief History of Ralph.” <https://www.humanlayer.dev/blog/brief-history-of-ralph>
- HumanLayer. (2025). “Advanced Context Engineering for Coding Agents (ACE-FCA).” <https://github.com/humanlayer/advanced-context-engineering-for-coding-agents>
- Malone, D. (2026). “Orchestrating AI Agents to Migrate a 3-Year-Old Codebase.” <https://www.dan-malone.com/blog/ralph-wiggum-orchestrating-ai-agents>
- Ronin Consulting. (2025). “Using the Ralph Wiggum Loop.” <https://www.ronin.consulting/artificial-intelligence/using-the-ralph-wiggum-loop/>
- Osmani, A. (2025). “How to Write a Good Spec for AI Agents.” <https://addyosmani.com/blog/good-spec/>
- Thoughtworks. (2025). “Spec-Driven Development.” Technology Radar. <https://www.thoughtworks.com/en-us/radar/techniques/spec-driven-development>
- METR. (2025). “Measuring the Impact of AI on Software Engineering Tasks.”

Chapter 9

Chapter 8: Context Management Strategies

9.1 The `malloc` Without `free`

The **memory management analogy** makes the problem concrete. In C, `malloc()` allocates memory and `free()` releases it. Context engineering has `malloc` — you can add messages, tool results, and context to the array. But there is no `free`. Once a message is in the array, it stays there for the rest of the session. (Technically you could remove messages from the array before sending, but most frameworks don't support this, and doing it naively breaks conversational coherence.)

This matters because in traditional programming, memory leaks are bugs. In LLM context management, “leaks” are the default behavior. Every tool call, every file read, every error message — they all accumulate. The context only grows.

9.1.1 Context Window vs. Traditional Memory

Operation	Traditional Memory	Context Window
Allocate	<code>malloc(size)</code>	Add a message to the array
Use	Read/write the pointer	Model attends to the message
Free	<code>free(ptr)</code>	Not available
Garbage collect	Automatic (in GC languages)	Not available
Leak	Bug — wastes RAM	Default behavior — wastes context
Out of memory	Process crashes	Model enters dumb zone

The three “memory management” primitives we DO have:

1. **Don't allocate:** Avoid putting things in context that aren't needed (use sub-agents — Chapter 5)
2. **Compact:** Summarize old messages to reduce their token count (lossy — see below)
3. **Reset:** Start a fresh context window with curated state (the Ralph Wiggum Loop — Chapter 7)

9.2 Why Compaction Is Dangerous

Compaction (also called context compression or summarization) replaces older messages in the array with a shorter summary. For example, 20 messages of conversation (5,000 tokens) might be compressed to a 500-token summary.

Current tools handle this in different ways. Claude Code automatically compresses older messages into a summary when context reaches ~80% of the window. ChatGPT uses a “memory” system that extracts key facts from conversations. LangChain offers `ConversationSummaryMemory` that summarizes older turns.

Understanding the mechanics reveals why this is fragile. Auto-compact typically fires at ~95% capacity (~167K of a 200K token window) — by the time it triggers, you're already deep in context pressure. `CLAUDE.md` (and similar injected instruction files) survive compaction because they're re-injected fresh from disk on every turn — they're the one thing that persists fully intact. Old file reads, grep results, and command outputs account for 60–80% of the content removed during compaction — these are high-token, low-signal messages once acted upon. A practitioner tip: run `/compact` manually with custom instructions (e.g., `/compact summarize only the to-do items and keep all file paths`) at strategic moments rather than letting auto-compact fire at the worst possible time.

The core danger is **non-deterministic eviction**. You can't control what the model considers important when summarizing. The model decides what to keep and what to discard. This is fine for casual conversation. It's dangerous for agent workloads where a specific instruction from 30 messages ago is still critical, a file path mentioned early in the conversation is needed later, an error pattern from the first test run needs to inform the fourth test run, or a constraint (“never modify the database schema”) was stated once and must persist.

Real failure scenarios illustrate the risk. **Lost constraints:** “Use only the existing API endpoints” gets compacted away, and the agent creates new endpoints. **Lost context:** A critical error message from an earlier tool call is summarized as “encountered some errors,” and the specific error information is gone. **Lost decisions:** “We decided to use approach B because approach A had race conditions” gets compacted to “using approach B,” so the next iteration doesn't know why and

might revisit approach A. **Merged context:** Information from separate tool calls gets merged in the summary, creating false associations.

The fundamental problem is that compaction is lossy compression performed by the same model that’s already struggling with context management. You’re asking a model that loses information in long contexts to decide what information is safe to discard. This is asking the fox to guard the henhouse. As Huntley puts it in the Ralph Wiggum Loop (Chapter 7): “Compaction is the devil.” Here’s why he’s right. The model performing the compression has no ground truth about what matters downstream — it’s making a bet, and every bet it loses destroys information you can never recover. The only safe compaction is the kind you control explicitly, with protected categories and priority tags (see Strategy 3 below).

9.3 Better Strategies

Given the dangers of compaction, what should you actually do?

9.3.1 Strategy 1: Prevention Over Cure (Sub-Agent Delegation)

The best strategy is to prevent context bloat in the first place. Delegate high-token operations to sub-agents (Chapter 5) and only receive summaries in the parent context. This is like avoiding memory allocation rather than trying to free it later.

9.3.2 Strategy 2: Fresh Context with Curated State (The Ralph Wiggum Loop)

When context accumulation is inevitable, reset deliberately. Write critical state to a persistent file (spec/plan), start a fresh context window, and load only what’s needed (Chapter 7). This is the closest thing to `free()` — you’re starting with a clean heap.

9.3.3 Strategy 3: Priority-Based Eviction

If you must compact, don’t let the model decide what to keep. You decide.

Protected categories (never evict) include the system prompt and constraints, the current task specification, active error states being debugged, and user-stated requirements and decisions.

Evictable categories (safe to summarize) include successful tool calls whose results have been acted on, exploratory reads that didn’t lead to action, and verbose output where only the conclusion matters.

A **counter-intuitive finding** from Manus (the AI agent company) is that failed actions should stay visible in context. Leaving wrong turns visible improves behavior — the model performs implicit

belief updates and avoids repeating the same mistakes. Evict successful tool output, but keep failure traces.

Observation masking (JetBrains Research, 2025) offers another approach: instead of summarizing entire turns, keep the agent’s reasoning and actions but replace older tool outputs with placeholder text. A rolling window of ~10 turns of full detail, with older observations masked. This achieved 50%+ cost reduction while actually *improving* solve rates by 2.6% — better than LLM summarization, which paradoxically made agents run 13–15% longer.

For **implementation**, maintain a priority tag on each message. When compaction is triggered, only compact messages tagged as evictable.

9.3.4 The Write/Select/Compress/Isolate Taxonomy (Lance Martin)

A useful framework for reasoning about context management strategies comes from Lance Martin’s “Context Engineering for Agents.” Every technique maps to one of four primitives:

Write means saving information outside the context window for later retrieval — scratchpads, memory files, persistent state stores. **Select** means retrieving only relevant information into context when needed — RAG, dynamic tool descriptions, filtered search results. **Compress** means retaining only essential tokens from what’s already in context — summarization, observation masking, trimming tool outputs. **Isolate** means splitting context across separate components so no single window bears the full load — sub-agents, sandboxed execution environments.

The strategies above map cleanly: sub-agent delegation is **Isolate**, the Ralph Wiggum Loop combines **Write** (persist state to file) and **Select** (load only what’s needed), priority-based eviction is **Compress**, and the structured stores below are **Write** + **Select**.

9.3.5 Strategy 4: Structured Context Stores (Emerging)

Rather than dumping everything into the linear messages array, use external structured stores.

RAG (Retrieval-Augmented Generation) works as “virtual memory”: store information outside the context window, retrieve it on demand. Like virtual memory paging in an OS — not everything needs to be in “RAM” (context) at once. **Semantic caching** takes this further by caching model responses keyed by semantic similarity of the input. If a similar question was answered recently, reuse the cached answer without re-reading source material. **Session state stores** are external key-value stores where agents can `write("auth_approach", "JWT with refresh tokens")` and later `read("auth_approach")`. This provides persistent memory that doesn’t consume context tokens until retrieved.

9.3.6 Strategy 5: The Context Manager Pattern

Instead of blindly appending to the messages array, **dynamically construct it on every turn**. Maintain state externally and assemble the prompt from current state each time.

Without a context manager (naive append):

```
[System prompt]
[Summary: "The user wanted a Python script. I wrote a function. It had a bug..."]
[Turn 47]
[Turn 48]
```

The model forgot the coding style guidelines from turn 2 — they didn’t survive summarization.

With a context manager (dynamic construction):

```
[System prompt + style guidelines]          (pinned - always included)
[Current file: auth.py]                    (working memory - active task)
[Latest request: "Fix the failing test"]    (current goal)
[Test output: 1 failure, truncated to key lines] (tool result - trimmed)
```

The context manager explicitly dropped 45 turns of history. It kept what matters: instructions, current state, and the immediate task. Token count is controlled. Critical context is preserved.

This pattern also directly solves the Lost-in-the-Middle problem from Chapter 2. Recall the U-shaped attention curve: models attend most strongly to tokens at the beginning (primacy) and end (recency) of the context window, while information buried in the middle gets the least attention. The Context Manager exploits this by pinning critical state — system prompts, style guidelines, constraints — to the top of the assembled context (primacy zone) and placing the current task and latest tool output at the bottom (recency zone). Everything in between is expendable working memory. Instead of hoping important information survives the middle of a bloated conversation log, you guarantee it lands where the model actually looks.

This is the difference between treating context as “a conversation log” and treating it as “a dynamically assembled working set.”

9.3.7 Strategy 6: Context-Aware Architecture Design

Design your agent system to minimize context pressure from the start. **Narrow tool outputs** means configuring tools to return minimal output — a test runner that returns only failures (not full pass/fail logs) saves thousands of tokens. **Pagination** means returning paginated results for large result sets (search, file listings) and letting the agent request more only if needed. **Structured over verbose** means returning JSON or structured data instead of human-readable narratives, since

structured data is easier to compress and easier for models to parse. **Time-bounded sessions** means setting a context budget alarm (e.g., at 50% utilization) that triggers the agent to wrap up or delegate remaining work.

9.4 Key Takeaways

- Context management has `malloc` but no `free`. The default is to leak.
- Compaction is lossy and non-deterministic — the model decides what to discard. Auto-compact fires at ~95% capacity; CLAUDE.md survives because it’s re-injected from disk.
- Prevention (sub-agents) beats cure (compaction). Observation masking beats LLM summarization.
- When you must compact, use priority-based eviction — protect critical context, keep failed actions visible.
- All strategies map to four primitives: Write, Select, Compress, Isolate.
- Emerging patterns (RAG, semantic caching, structured stores) offer alternatives to the linear messages array.

9.5 References

- Kernighan, B. & Ritchie, D. (1988). *The C Programming Language*, 2nd Ed. The `malloc/free` analogy originates here.
- Lewis, P., et al. (2020). “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.” arXiv:2005.11401. <https://arxiv.org/abs/2005.11401>
- Packer, C., et al. (2023). “MemGPT: Towards LLMs as Operating Systems.” arXiv:2310.08560. <https://arxiv.org/abs/2310.08560>
- Anthropic. “Compaction.” <https://platform.claude.com/docs/en/build-with-claude/compaction>
- Anthropic. “Effective Context Engineering for AI Agents.” <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>
- Manus. “Context Engineering for AI Agents: Lessons from Building Manus.” <https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus>
- JetBrains Research. (2025). “Efficient Context Management for Coding Agents.” <https://blog.jetbrains.com/research/2025/12/efficient-context-management/>
- Martin, L. (2025). “Context Engineering for Agents.” https://rlancemartin.github.io/2025/06/23/context_engineering/

Chapter 10

Summary

Context engineering is the discipline of managing the scarcest resource in AI systems: the context window. This book has established that:

1. **Tokens are the currency** — everything has a cost measured in tokens (Chapter 1)
2. **Context windows are smaller than advertised** — design for the smart zone, not the marketing number (Chapter 2)
3. **The messages array has a fixed-cost structure** — system prompt, harness, project context, and tools consume tokens before your conversation even begins (Chapter 3)
4. **Tool calls are memory allocations** — they grow context permanently within a session (Chapter 4)
5. **Sub-agents are context isolators** — delegate reads to sub-agents, keep writes centralized (Chapter 5)
6. **Message passing is the architecture** — design explicit protocols between context windows (Chapter 6)
7. **Fresh context beats stale context** — the Ralph Wiggum Loop (Requirements → Planning → Building) uses the filesystem as memory and the context window as disposable compute (Chapter 7)
8. **Prevention beats compaction** — avoid filling context rather than trying to compress it after the fact (Chapter 8)

The central insight: **treat the context window as you would treat memory in a systems programming language** — as a finite, precious resource that requires deliberate management.

Chapter 11

Publication History

11.1 Credits

This book was written by [Claude](#) (Anthropic), directed and edited by a human author across multiple sessions using aggressive parallelism — research agents, drafting agents, and review agents running concurrently.

The outline originates from [Latent Patterns](#). Content was developed through deep research into practitioner experiences, academic papers, and production systems. Over 50 sources are cited throughout.

11.2 Timeline

March 2026 — Initial publication. 8 chapters covering tokens, context windows, the messages array, tool calling, sub-agents, message passing, the Ralph Wiggum Loop, and context management strategies.

Structural review — Chapters 5-7 reordered based on external feedback to follow a micro-to-macro progression: Sub-Agents (isolate heavy calls) → Message Passing (structure communication) → Ralph Wiggum Loop (reset the whole session). Crash-only software framing added to the Ralph Wiggum Loop. Cross-references tightened: Lost-in-the-Middle linked to Context Manager pattern, compaction foreshadowing, read/write parallelism linked to brownfield challenges.

11.3 Design Decisions

- **Prose over bullets** — textbook-style flowing paragraphs, not slide-deck lists

- **Concrete numbers** — every claim backed by specific data (RULER scores, token counts, dollar costs)
- **Practitioner depth** — failure modes, criticisms, and real-world numbers alongside theory
- **Honest treatment** — includes limitations (METR complexity ceiling, code quality concerns, cost)

11.4 License

This work is licensed under [CC BY-ND 4.0](#). You are free to share it with attribution.