Day 2: System Prompts & CLAUDE.md

The Foundation of Context Engineering

Isham Rashik · AI Engineer
Engineering AI with Clarity
NLP • Computer Vision • Fine-Tuning • RAG • Agentic AI

Version: v1.0 · April 25, 2026

Acknowledgment

I am deeply grateful to Dr. Sreedath Panat and **Vizuara Technologies** for creating and generously sharing the *Context Engineering* course. These notes would not exist without his exceptional teaching, clear explanations, and dedication to making cutting-edge AI concepts accessible to everyone.

Thank you, Dr. Panat, for the time, effort, and passion you have poured into this course. Your work has been instrumental in shaping my understanding of context engineering.

→ Watch the full playlist on YouTube

Contents

1 Prerequisites

- Familiarity with the six layers of context from the companion Day 1 guide - Introduction to LLM Context Engineering
- Basic understanding of LLM API calls (sending a prompt, receiving a completion)
- Markdown syntax (headings, subheadings, lists)
- Familiarity with Claude Desktop or Claude Code

2 Overview

This guide dives deep into **system prompts**: the foundational layer of context provided to large language models. The primary vehicle for system prompts in Claude Code is the **CLAUDE.md** file. It covers the five components of a well-structured system prompt (identity, rules, format, knowledge, tools), the "right altitude" principle for writing instructions, iterative construction methodology, selective retrieval of sections, the file hierarchy (`CLAUDE.md` , `AGENTS.md` , `SKILL.md` , `SOUL.md`), few-shot example strategies, and the coding agent landscape including Claude Code, Google Antigravity, Cursor, and Copilot.

Building on the six-layer context model from the companion guide on context window fundamentals, this guide focuses entirely on Layer 1: the system prompt.

3 Recap: The Six Layers of Context

The six elements that constitute the context provided to an LLM are:

1. **System Prompt**: rules, guardrails, identity (`CLAUDE.md` / `AGENTS.md`)
2. **User Instructions**: the direct prompt or query from the user
3. **Tools**: functions, APIs, libraries accessible to the model (MCP)
4. **Memory / State**: persistent knowledge (preferences, historical patterns)
5. **RAG**: dynamically fetched data from knowledge bases
6. **Conversation History**: the current session's message exchanges

These six elements are not listed in any particular priority order. The *order in which they appear in the context window* is what matters.

3.1 Memory vs. Conversation History

A common point of confusion: memory and conversation history are **not** the same thing.

Table 1: Memory vs. conversation history

Aspect	Conversation History (Session Memory)	Memory / State (Persistent Memory)
Scope	Current conversation thread only	Across all conversations and sessions
Lifespan	Exists for duration of one session	Persists for weeks, months, or indefinitely
Content	The 25 email exchanges in a thread	Your email preferences over the past year
Example	"What was said 3 messages ago?"	"This user prefers formal tone in technical emails"
Storage	Context window	Memory markdown files, databases, knowledge bases

Concrete example: An email agent responding to a thread. The 25 prior exchanges in that specific thread form the **conversation history**. How the user has replied to technical vs. non-technical emails over the past year, distilled from 5,000–10,000 email exchanges, forms the **persistent memory/state**.

3.2 Context Ordering and the Lost-in-the-Middle Effect

The context window has a well-documented attention bias. LLMs give the most prominence to the **beginning** and **end** of the context window. Content in the middle receives slightly less attention: this is the “lost in the middle” effect.

- **System prompt** (rules, guardrails) goes at the **beginning**: highest priority
- **User prompt** goes at the **end**: also high priority
- Everything else (RAG output, tool results, memory) sits in the **middle**

> 🔥 **Practical Guideline**
>
> Even if an LLM supports 1–2 million tokens as its context window, try to keep your actual context length to **50,000–100,000 tokens**. Larger contexts amplify the lost-in-the-middle effect.

With the six-layer model refreshed, this guide now focuses entirely on Layer 1: the system prompt. What goes into it, how to structure it, and how to engineer it for maximum effectiveness.

4 Anatomy of a System Prompt (CLAUDE.md)

A `CLAUDE.md` file typically contains five components. These form the skeleton of any well-structured system prompt.

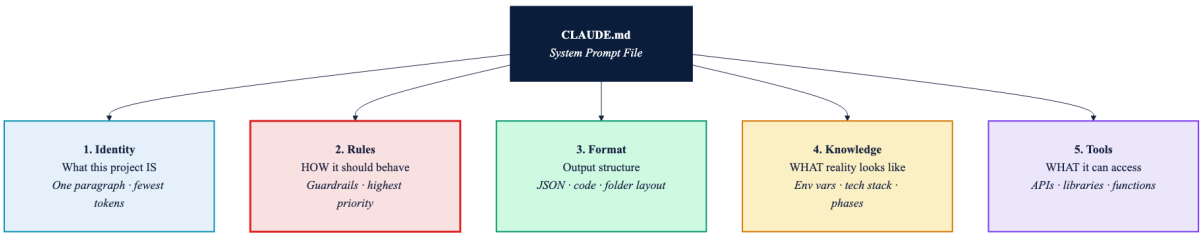

Figure 1: Five components of CLAUDE.md

4.1 Identity

Identity is the first thing written in `CLAUDE.md` and often the least important in terms of behavioral impact. It describes **what the project is**: not in anthropomorphic terms ("you are a senior code reviewer"), but in functional terms.

Good Examples

- This is an email responding agent
- This is a personal digital clone for responding to Slack messages
- This is a website that automatically updates when new developments happen in the AI space

Key Point

Identity does not mean "who the LLM is." It means what the thing you are building is meant to do. Avoid giving human-like personas unless specifically needed.

Token characteristics: Identity is typically one paragraph: it consumes the fewest tokens of all five components. If forced to cut sections, identity is the **safest to remove first** because the model can often infer its role from the rules, examples, and conversation context.

If Token budget is tight — what to cut first? → Cut Identity — The model can infer its role from rules, examples, and conversation. Format can default to natural language. Rules must never be cut first — they are the last line of defence.

Identity orients the model without constraining it. The real behavioral control lives in the next component.

4.2 Rules

Rules define **how** the model should or should not behave. These are the guardrails — the component that most directly determines whether the system produces correct, safe, and consistent outputs.

Examples

- Never respond about refund policy without consulting the policy document
- Never use `camelCase` - use `snake_case`
- Never store student scores with more than two decimal places
- If a customer asks about returns, consult the return policy document before responding

Priority: Rules are the **highest priority** component. Even under extreme token constraints, rules should be the last component removed. Rules must be loaded into the context for every single API call because guardrails should never be broken.

Rules vs. Knowledge distinction:

- **Rules** = how to do something (behavioral directives).
- **Knowledge** = what something is or what reality looks like (factual descriptions).

Rule of Thumb

If it says “never do X” or “always do Y”: it is a rule. If it describes what exists or how things are structured: it is knowledge.

Rules govern behavior; they do not govern output shape. That is what the next component handles.

4.3 Format

Format specifies **how the LLM should structure its output** — independently of what it says or how it behaves.

This includes:

- Output format (Python code, JSON, natural language, CSV)
- Project folder structure (directory tree showing where files go)
- Development commands (`npm run dev` , `npm run build`)

Example: A Project Structure Definition

```
src/
├── calculator.py  # calculate_final_grade function
├── models.py      # data classes
├── utils/
└── tests/
```

By providing format specifications in `CLAUDE.md`, you reinforce the structure you want the project to follow and ensure the LLM knows where files are located. With behavior and output shape defined, the model still needs factual grounding about the application it is operating within.

4.4 Knowledge

Knowledge describes **what the reality of the application is**: factual information the model needs to do its job — not instructions on how to behave, but the facts it must know to apply those instructions correctly.

Examples

- Environment variables and how they are structured
- Development phases (Phase 1: MVP, Phase 2: Multi-user)
- Tech stack details (Python, ChromaDB, LangChain)
- How the project overview describes the application

Knowledge fills the gap between what the model knows in general and what it needs to know specifically about this project. The final component extends the model's reach beyond static knowledge to active capabilities.

4.5 Tools

Tools define **what the model can access**: APIs, libraries, packages, and functions the model is permitted and expected to use.

Examples of Tool Declarations

- Use Claude API for LLM calls
- ChromaDB for vector storage
- LangChain or LlamaIndex for orchestration
- `CSV.writer` for export operations

Tools define the model's reach — what it can actually do beyond generating text. Together, the five components form a complete system prompt. In practice, however, their boundaries are rarely clean.

4.6 Blurred Boundaries Between Components

The boundaries between components are **fluid, not rigid**:

Table 2: Blurred boundaries between system prompt components

Boundary	Example	Why It Is Ambiguous
Rules ↔ Knowledge	"Start with cosine similarity"	Is this a rule (how to do retrieval) or knowledge (what method to use)?
Knowledge ↔ Tools	Tech stack section listing Python, ChromaDB	Defines both what tools to use AND knowledge about the stack
Rules ↔ Format	Development commands (npm run dev)	Specifies both how to run the project AND the format of operations
Knowledge ↔ Format	Project folder structure	Describes what exists AND specifies output structure

Info

Context engineering is **not physics**: it is more like a collection of thumb rules that evolved from experimentation by large companies and communities. The boundaries between knowledge, rules, and format are deliberately soft.

Now that the five components are clear, the next question is: how specific or general should instructions within each component be? Anthropic offers a guiding principle.

5 Right Altitude Principle

Every instruction in `CLAUDE.md` has a cost: it consumes tokens, adds noise, and competes with every other rule for the model's attention. Anthropic's documented guidance [1] is direct — keep `CLAUDE.md` short and human-readable, and apply a single test to every line:

> **If Would removing this line cause Claude to make mistakes?** → **Keep it** — It encodes something the model cannot infer on its own
> **If Would removing this line make no difference?** → **Cut it** — Bloated CLAUDE.md files cause Claude to ignore your actual instructions

Figure 2: The right altitude: too high and you see nothing useful, too low and you crash into every detail

This test maps directly onto two failure modes: instructions too vague to act on, and instructions so specific they break the moment the world changes.

The two failure modes — too vague and too brittle — are best understood side by side with the right altitude that sits between them.

Table 3: Right altitude examples: specific enough to guide, flexible enough for heuristics

Domain	Too High (Vague)	Too Low (Brittle)	Right Altitude
Pricing	Be helpful with pricing questions	Reply: Plans start at $9.99/month	Check the pricing database first. If no exact match, suggest the closest plan and explain the difference.
Errors	Handle errors gracefully	If get_orders returns 404, say 'Order not found'	When a tool call fails, explain what happened in plain language, suggest one alternative action, and offer to escalate to a human.
Tone	Be professional	Always start with 'Thank you for reaching out!' End with 'Is there anything else I can help with today?'	Use a warm but professional tone. Mirror the user's formality level. Avoid jargon unless the user uses it first.
Code style	Write clean code	All variables must use snake_case. All functions must have exactly one return statement.	Follow PEP 8. Use type hints on all function signatures. Functions should be under 30 lines. Use descriptive names, no single-letter variables except loop counters.

> ⚠️ **Common Mistake**
>
> Saying “do not hallucinate” is the most vague thing you can ever write in a system prompt. It provides zero actionable guidance to the model.

> ⚡ **Why Brittle Instructions Fail**
>
> CLAUDE.md has a lifespan of **weeks to months**. Hardcoding specific prices, error messages, or API responses means the file becomes stale the moment any detail changes.

5.1 What to Include vs. Exclude

Anthropic’s official documentation [1] provides a concrete include/exclude breakdown that operationalizes the right altitude principle:

Table 4: Anthropic's official include/exclude guidance for CLAUDE.md

Include ✓	Exclude ✗
Bash commands Claude cannot guess	Anything Claude can figure out by reading code
Code style rules that differ from defaults	Standard language conventions Claude already knows
Testing instructions and preferred test runners	Detailed API documentation (link to docs instead)
Repository etiquette: branch naming, PR conventions	Information that changes frequently
Architectural decisions specific to your project	Long explanations or tutorials
Developer environment quirks, required env vars	File-by-file descriptions of the codebase
Common gotchas and non-obvious behaviors	Self-evident practices like "write clean code"

> 🔥 **Treat CLAUDE.md Like Code**
>
> Review it when things go wrong, prune it regularly, and test changes by observing whether Claude's behaviour actually shifts. If Claude keeps violating a rule despite it being written down, the file is probably too long and the rule is getting lost in the noise. Add emphasis — **IMPORTANT** or **YOU MUST** — for rules that must not be ignored.

The right altitude principle applies to every component of CLAUDE.md. With that clarity on what to write and what to cut, the next question is how to actually build the file.

6 System Prompt Formatting: XML vs. Markdown

The first decision is structural: how should a system prompt be marked up? Two options are widely used in practice.

Table 5: XML vs. Markdown formatting for system prompts

Format	Structure	Model Preference
XML	<identity>...</identity> <rules>...</rules> tags	Claude reportedly prefers XML based on documentation
Markdown	# Identity ## Rules with headings/subheadings	More universal, works well with all models

> 🔥 **Recommendation**
>
> Use **Markdown** for `CLAUDE.md`. The difference in output quality between XML and Markdown is **not perceivable** in practice. Markdown is also the natural choice if you ever write system prompts for other platforms (Cursor, Copilot, Windsurf), since those tools use their own Markdown-based config files — not `CLAUDE.md` itself, which is **Claude Code only**. The heading hierarchy (#, ##, ###) provides the same logical structure as XML tags.

Markdown heading hierarchy enables section-based retrieval:

- `#` = top-level section
- `##` = subsection
- `###` = sub-subsection

Each heading creates a natural chunk boundary that can be independently retrieved. This property becomes critical as `CLAUDE.md` files grow larger — Markdown structure is what makes selective retrieval possible at all.

With format decided, the next question is how to fill it. Writing everything at once leads to a predictable trap.

7 Iterative Construction: Start Minimal, Then Add

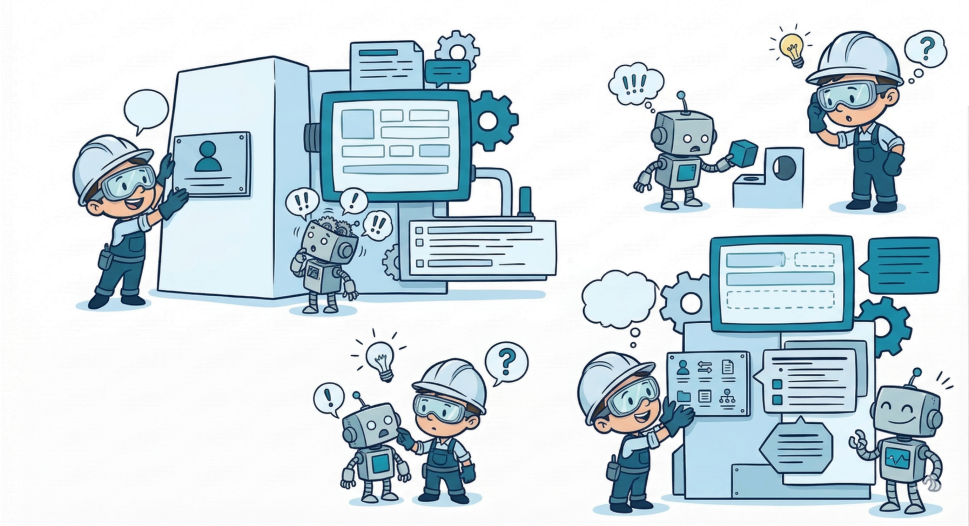

Figure 3: Building a system prompt: start with identity, observe failures, add one rule at a time

The temptation when writing `CLAUDE.md` is to define every rule upfront. Anthropic's recommendation is the opposite: start with the minimum viable system prompt and add rules only when observed failures demand them.

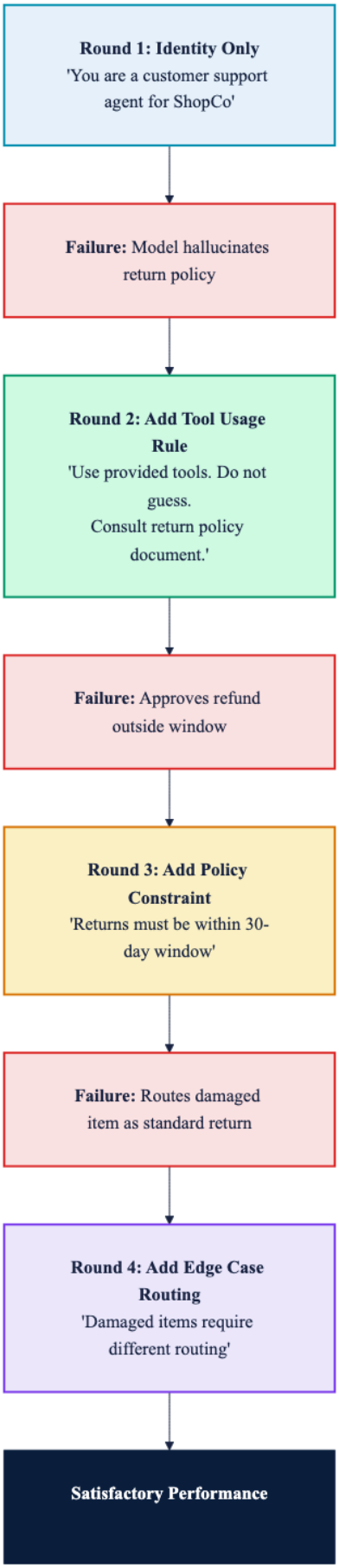

Figure 4: Iterative system prompt construction: each failure reveals the next rule to add

1. **Round 1: Minimal Identity (Score: 3/10)** — "You are a customer support agent for ShopCo." Run on real tasks. Failure: model hallucinates return policy.

2. **Round 2: Add Tool Usage Rule (Score: 5/10)** — Fix: "Use provided tools for actions. Do not guess." +Anti-hallucination. Failure: approves refund outside window.

3. **Round 3: Add Policy Constraint (Score: 7/10)** — Fix: "Returns must be within 30-day window." +Empathy-first, de-escalation. Failure: routes damaged item as standard return.

4. **Round 4: Add Edge Case Routing (Score: 9/10)** — Fix: "Damaged items require different routing." +Edge cases, routing rules. Result: all tests pass within ~200 token budget.

Each failure reveals a missing rule or piece of knowledge. The scores progress from 3/10 to 9/10 over four rounds, with the total prompt growing from 30 tokens to 200 tokens. Every rule earns its place by fixing a real failure, not through speculation. This discipline also protects against a deeper structural problem that emerges when rules are written in bulk.

7.1 Avoiding Contradictions

The primary danger of writing a comprehensive system prompt in one shot is **self-contradiction**.

⚡ **How Contradictions Happen**

When defining 10–20 rules at once, some rules may conflict. The model then picks arbitrarily, and picks wrong.

- Rule: Refund window must be within 30 days
- User preference: If the case is legitimate, always provide the refund
- **Conflict:** What if the case is legitimate but it is day 45? The model cannot satisfy both rules simultaneously.

If Building system prompt from scratch → **Iterative build (preferred)** — Add rules one at a time, test after each addition
If Writing full CLAUDE.md at once → **Manual review (minimum)** — Read through carefully to check for contradictions before deploying

> **Memorize**
>
> The least you can do is either iterate and build, or build the `CLAUDE.md` file and read through it once properly to catch contradictions.

Iterative construction naturally limits contradictions because each rule is added in context, with the previous rules visible. With a clean, conflict-free system prompt taking shape, the remaining question is practical: how do you get from raw project requirements to a well-structured `CLAUDE.md` in the first place?

8 Construction Workflow: From Raw Ideas to CLAUDE.md

Capturing requirements is often the hardest part. In practice, project knowledge lives in developers' heads and in scattered notes — not in a form that can be directly dropped into a system prompt. A three-step workflow bridges that gap.

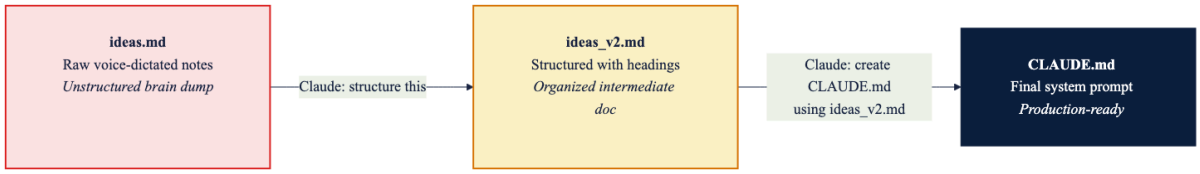

Figure 5: Three-step workflow: unstructured brain dump → structured intermediate → final CLAUDE.md

1. **ideas.md — Brain Dump** — Use a voice-to-text tool (e.g., Wispr Flow) to rapidly dump all requirements. This file is intentionally unstructured — a brain dump.

2. **ideas_v2.md — Structured Intermediate** — Ask Claude to restructure the raw ideas into a well-organized markdown document with headings and subheadings. This is NOT the CLAUDE.md yet.

3. **CLAUDE.md — Final System Prompt** — Ask Claude to create the CLAUDE.md file using ideas_v2.md as the source. The CLAUDE.md references ideas_v2.md for full project context.

> **? Why not go directly from ideas.md to CLAUDE.md?**
>
> The unstructured nature of ideas.md would produce a poorly structured CLAUDE.md. The intermediate step ensures proper organization before the system prompt is generated.

The result is a `CLAUDE.md` that is well-structured, grounded in actual project requirements, and built without contradiction. Once created, it is meant to stay that way.

8.1 Lifespan and Stability

Unlike most project files, `CLAUDE.md` is not meant to evolve continuously. **Its lifespan is weeks to months.**

- It should **not change** once finalized
- The basic ideas and rules it encodes should remain constant throughout the project lifecycle
- If something needs to change frequently, it probably belongs in a different file (`SKILL.md` , feedback files, etc.)

This stability is intentional. `CLAUDE.md` is the foundation; other files absorb day-to-day churn. Keeping it stable means the model always has a reliable ground truth to reason from — but that ground truth must also stay within manageable bounds.

8.2 Size Guidelines

A stable `CLAUDE.md` is not necessarily a large one. Size matters because token budget is finite, and an oversized file strains retrieval accuracy.

Table 6: CLAUDE.md size guidelines

Metric	Recommendation
Lines	200--500 lines (not more)
Words	~1,000--3,000 words
Tokens	~1,000--3,000 tokens
Maximum before concern	5,000+ lines is too large: selective retrieval becomes critical

> **Info**
>
> `CLAUDE.md` is **not entirely loaded all at once**. The most important parts (especially rules) are always loaded. Remaining sections are loaded on demand using keyword matching or RAG.

> **What if your system prompt hits 5,000+ tokens?**
>
> Split it into persistent memory and dynamic RAG-based retrieval. Keep rules and identity in the always-loaded persistent layer. Move knowledge sections and few-shot examples into retrievable chunks — only what is relevant to the current task gets loaded. This is not a workaround; it is the intended architecture.

Keeping `CLAUDE.md` within these bounds is not about brevity for its own sake — it is about making selective retrieval reliable. As the file grows, the next mechanism becomes essential: choosing which sections are even loaded for a given task.

9 Selective Section Retrieval from CLAUDE.md

9.1 How Retrieval Works

Not every section of `CLAUDE.md` is relevant to every API call. The system selectively retrieves sections based on the current task.

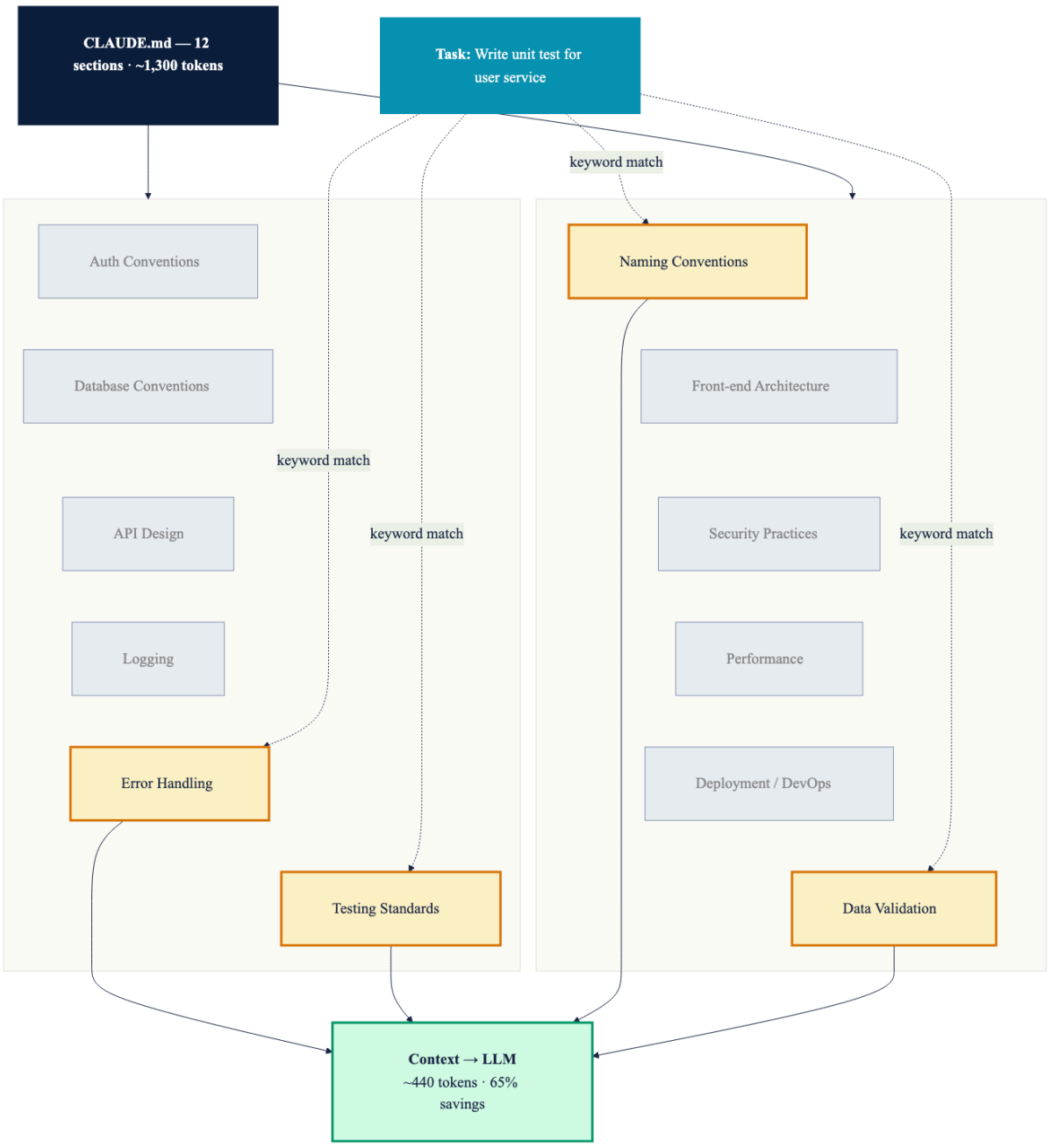

Figure 6: Selective retrieval: only the 4 task-relevant sections (highlighted) are passed to the LLM

Example: A `CLAUDE.md` with 12 sections totaling ~1,300 tokens. When the task is "write unit test for user service," only 4 relevant sections are retrieved (testing standards, error handling, naming conventions, data validation): consuming ~440 tokens instead of the full 1,300.

Table 7: Selective retrieval saves 65% of tokens compared to loading the full CLAUDE.md

Task	Sections Retrieved	Tokens Used
Unit test for user service	Testing Standards, Error Handling, Naming Conventions, Data Validation	~440
React form component	Front-end Architecture, Error Handling, API Design, Data Validation	~440

9.2 RAG-Based vs. Keyword-Based Retrieval

Table 8: RAG-based vs. keyword-based retrieval for CLAUDE.md sections

Method	How It Works	When Used
Keyword Matching	Match keywords in user query against section headings/content	Default in Claude Code; simpler and faster
RAG	Chunk CLAUDE.md → embed chunks → store in vector DB → compare query vector with chunk vectors → retrieve top-K	When explicitly configured; better for large/complex files

> 🔥 **Claude Code**
>
> In Claude Code, section retrieval is handled **automatically by the coding agent** [1]. You do not need to implement retrieval yourself. It is mostly done via keyword matching, not full RAG.

Implication for Markdown structure: Headings (`#`, `##`, `###`) create natural chunk boundaries. Each section under a heading becomes a retrievable unit. This is why Markdown structure matters: it directly enables selective retrieval.

The concept of selective retrieval raises a question about how different configuration files relate to each other. `CLAUDE.md` is not the only file in the system.

10 File Hierarchy: CLAUDE.md, AGENTS.md, and SKILL.md

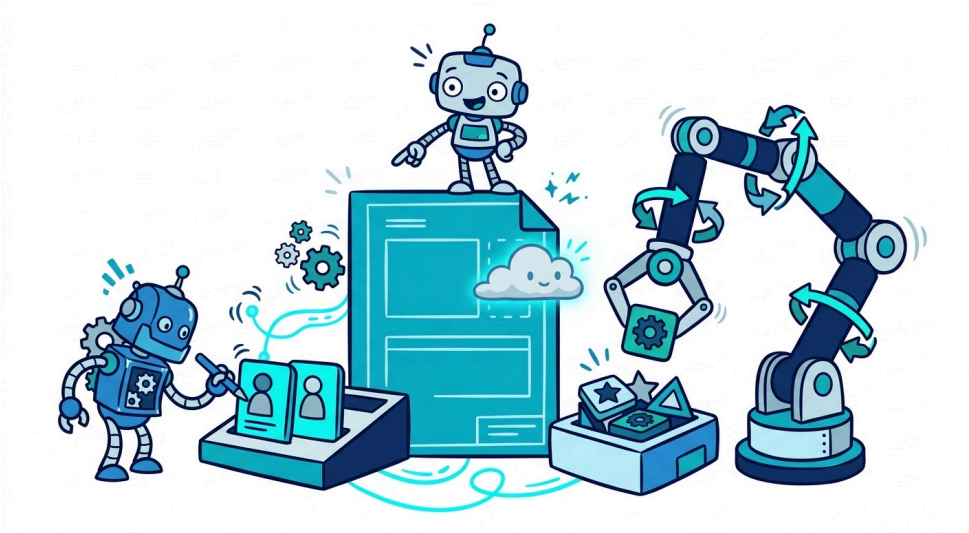

Figure 7: The configuration file family: CLAUDE.md sets the foundation, AGENTS.md defines the workers, SKILL.md teaches tricks, SOUL.md gives personality

10.1 CLAUDE.md - Four Scoping Levels

`CLAUDE.md` files can live at **four levels** [2], each with a different scope:

Table 9: CLAUDE.md scoping levels

Location	Scope	Loading Behavior
~/.claude/CLAUDE.md	All sessions, every project	Always loaded at launch
./CLAUDE.md (project root)	Current project only	Always loaded at launch; check into git to share with team
Parent directories	Inherited by all children	Loaded in full at launch (useful for monorepos: root/CLAUDE.md + root/api/CLAUDE.md)
Child directories	Only when working in that directory	Loaded on demand when Claude reads files in that subdirectory

`CLAUDE.md` also supports importing other files using `@path/to/import` syntax, for example: `@README.md`, `@docs/git-instructions.md`, or `@~/.claude/my-project-instructions.md`.

Strategy: If a preference is universal (web development patterns, coding standards), put it in the global `CLAUDE.md`. If it is project-specific (writing style for one product), put it in the project `CLAUDE.md`. Use child-level `CLAUDE.md` files for subdirectory-specific rules in monorepos.

10.2 AGENTS.md - Sub-Agent Definitions

`AGENTS.md` is an open standard **released by OpenAI** in August 2025 [3] and now governed by the Linux Foundation's **Agentic AI Foundation (AAIF)** [4]. It has been adopted by over 60,000 open-source projects and supported by major agent frameworks including Codex, Cursor, Devin, Gemini CLI, GitHub Copilot, Jules, and VS Code. It defines what sub-agents do.

Key differences from `CLAUDE.md`:

- `AGENTS.md` has much more **emphasis on specific tools** because agents are designed to access tools and execute functions
- `AGENTS.md` files can be **nested in subdirectories** to define a hierarchy of agents
- `CLAUDE.md` defines the overall project; `AGENTS.md` defines what individual agents within the project do

Research Paper Writing System

A root `AGENTS.md` orchestrates the entire research activity. Sub-agents for writing, coding, and analysis each have their own `AGENTS.md` with specific instructions. The coding agent further delegates to a RunPod code runner with its own `AGENTS.md`: forming a nested hierarchy.

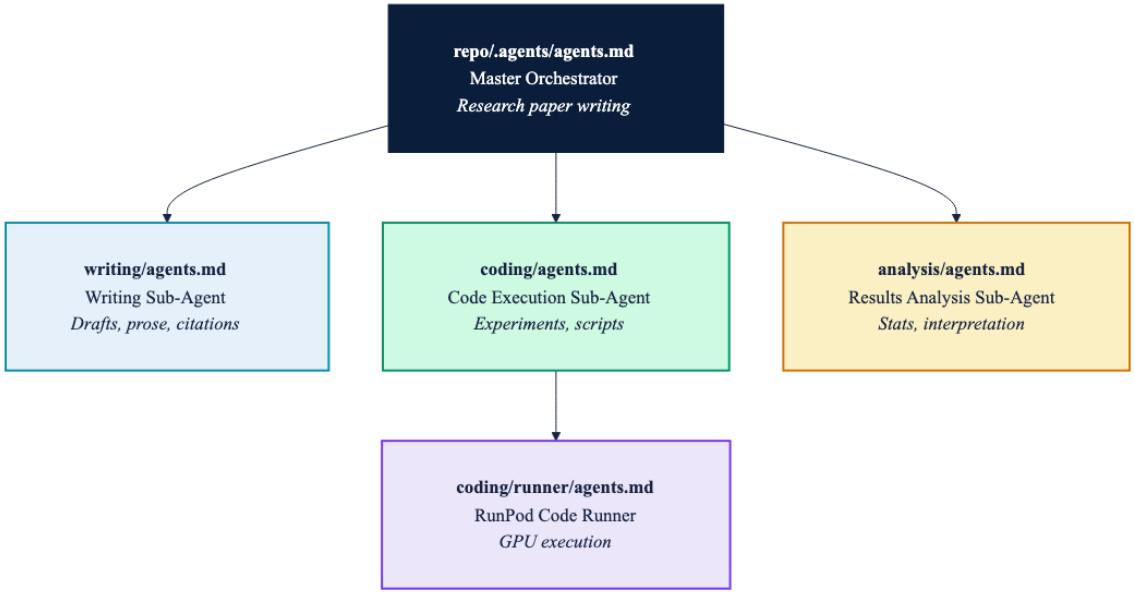

Figure 8: Nested AGENTS.md hierarchy for a research paper writing system

10.3 SKILL.md - Capability Definitions

`SKILL.md` files define **specific skills** the model should have for a domain.

Examples

- Front-end development preferences ("use Next.js, thin fonts, modern styled websites")
- API design patterns
- Writing style preferences

Lifespan: Hours to days. `SKILL.md` files can be heavily modified as more features or specifications are added.

Sources for skills:

- Write them yourself in natural language
- **skills.sh**: a marketplace for community-created skills
- **Everything Claude Code** (GitHub, ~69K stars): a plug-in that installs a comprehensive set of skills and agents

10.4 SOUL.md

`SOUL.md` is a special file (not Claude Code specific) that defines the **personality and character** of an agent.

> **Quote**
>
> The difference between your output when you have a `SOUL.md` versus when you don't have a `SOUL.md` is **night and day**.
>
> — Course Instructor - Dr.Sreedath Panat

- In Claude Code, the `CLAUDE.md` itself serves this function
- In other frameworks (OpenClaw, etc.), a dedicated `SOUL.md` is recommended
- Contains detailed specifications of what the agent's personality, tone, and behavioral characteristics should be

10.5 Hierarchy Resolution and Override Rules

When multiple `AGENTS.md` files exist in a nested hierarchy:

1. **All** `AGENTS.md` files are loaded and considered
2. If there is a **conflict** between a parent and child `AGENTS.md`, the **deeper (more specific) file's rules override** the parent's rules
3. Non-conflicting rules from all levels are **merged and used together**

> **CSS Analogy**
>
> Like CSS specificity: more specific selectors override general ones, but non-conflicting styles from all levels apply.

> **How Codex Builds an Instruction Chain**

```
repo-root/
  AGENTS.md        <-- Read first (project-wide rules)
  src/
    AGENTS.md      <-- Read second (src-specific rules)
    api/
      AGENTS.md    <-- Read third (api-specific rules)

Codex assembles: root rules + src rules + api rules
= complete instruction chain for a file in src/api/
```

A question naturally arises from the `AGENTS.md` discussion: do the sub-agents defined in `AGENTS.md` share context with their parent? The answer has major architectural implications.

11 Sub-Agents and Context Window Isolation

A critical architectural question: do sub-agents share their parent's context window?

Answer: No. Each sub-agent has its own context window.

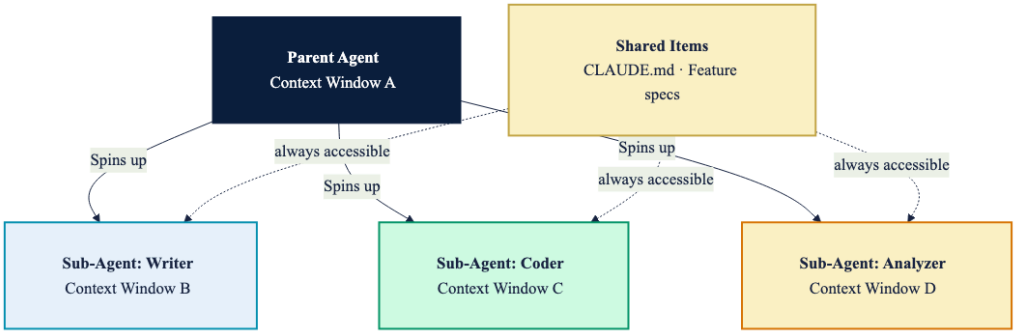

Figure 9: Each sub-agent holds a separate context window; CLAUDE.md and feature specs are shared via dotted links

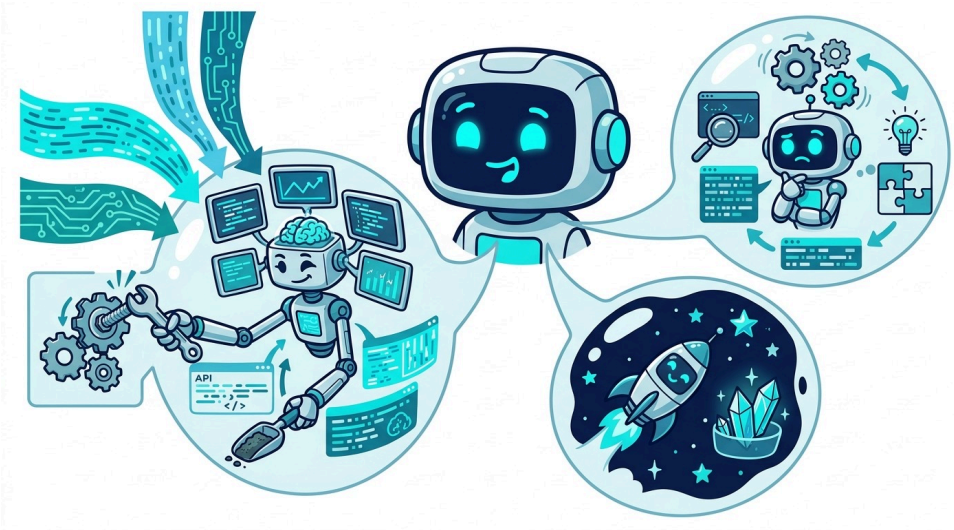

Figure 10: Each sub-agent operates in its own context bubble: no cross-contamination

Rationale:

- An agent is defined by its ability to make **its own API call** to the LLM
- Each agent handles **its own context window** so its actions are not polluted by what is happening globally
- A paper-writing agent has no reason to share context with a coding agent
- Sub-agents make **separate API calls** and produce outputs based purely on their own context

What IS shared: Certain items like `CLAUDE.md` are accessible to all agents. Feature specifications may also be shared. But the context window itself is isolated.

Understanding context isolation leads to a broader question: how long does each type of context information last? The answer reveals a hierarchy of lifespans.

12 Context Lifespan Hierarchy

Different elements of context have vastly different lifespans:

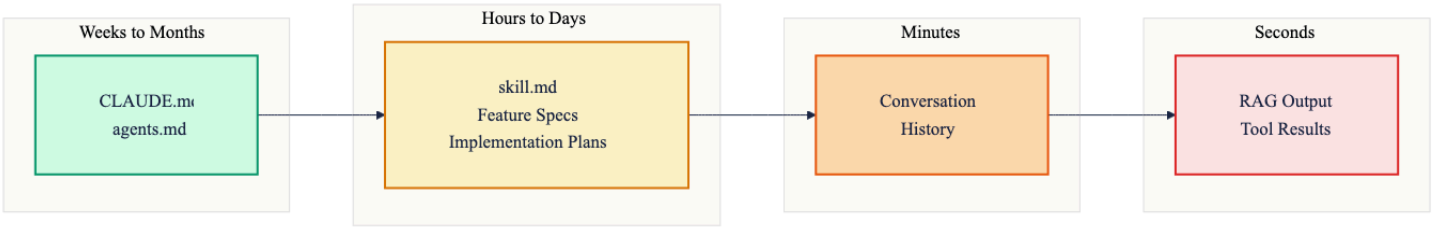

Figure 11: Context lifespan hierarchy: foundation documents live for months; tool results live for seconds

The context stack operates like an operating system: Layer 1 is the kernel (always running), Layer 5 is the application (comes and goes).

Table 10: Context lifespan hierarchy

Layer	Content	Lifespan	When Loaded
Layer 1: CLAUDE.md / AGENTS.md	Project rules, conventions, test requirements	Weeks to months	Automatically, every interaction
Layer 2: Feature Spec (INITIAL.md)	What to build, edge cases, links to docs	Hours to days	When the developer starts the feature
Layer 3: Implementation Plan (PRP)	Step-by-step blueprint with validation checkpoints	Hours to days	Generated once, referenced throughout
Layer 4: Examples Folder	Sample code, few-shot pairs, reference implementations	Days to weeks	Loaded when the agent needs a pattern reference
Layer 5: Dynamic Tool Results	RAG output, API responses, test output, linter output	Seconds	Generated fresh at each step
RAG output / Tool results	Seconds	Extremely dynamic; lives only for the current API call cycle	

The more foundational a piece of context is, the longer it should live and the more carefully it should be engineered. `CLAUDE.md` is the longest-lived and therefore deserves the most deliberate design.

🔥 **Fixing Context Rot**

When chatbot performance degrades over a long session, two interventions work: periodically **reinject** key system prompt rules mid-conversation, or add targeted few-shot examples that directly address the observed degradation. For example, adding a rule like "never refer to links published before November 2025" can immediately correct drift seen in production agents.

With the architecture and lifespan model established, a powerful technique for improving system prompt effectiveness remains: few-shot examples.

13 Few-Shot Examples in System Prompts

Providing examples in the system prompt significantly improves output quality. There are three types of examples based on task complexity.

💡 **Examples Beat Abstract Instructions**

When a system prompt says "be formal and professional" but the few-shot examples demonstrate a casual, friendly tone, the model follows the **examples** — not the instruction. Concrete demonstrations of desired behaviour carry higher weight than abstract directives. This means poorly chosen examples can silently override your written rules.

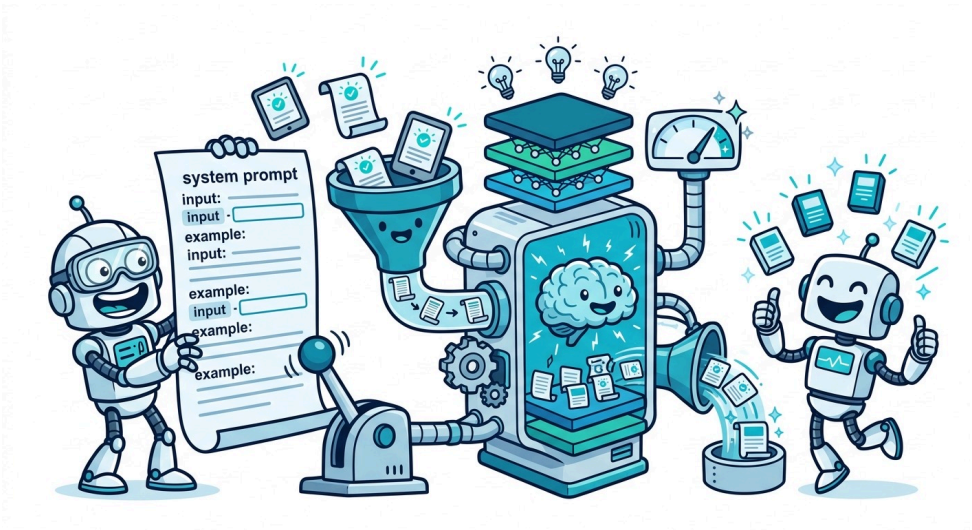

Figure 12: Few-shot examples: showing the model what you want is worth a thousand rules

13.1 Classification Tasks: Input/Output Pairs

The simplest case: the model needs to sort inputs into predefined categories. A few input/output pairs are enough to teach the mapping.

Input/Output Pair Examples — Classification

Example 1: [email about meeting invitation] → Primary Example 2: [email about LinkedIn notification] → Social Example 3: [email about software update] → Updates

Provide **up to 3 diverse examples**: one per category if possible. Classification examples are compact and cheap, making them the easiest win in any system prompt.

13.2 Reasoning Tasks: Chain-of-Thought Examples

When tasks require policy application, multi-step reasoning, or judgment, showing the input and output alone is not enough. The model needs to see **how to reason through** the problem.

Chain-of-Thought Example — Reasoning Task

Example: Input: "Customer ordered laptop 45 days ago, wants return" Reasoning: 45 days > 15-day return window → not eligible Output: "Return not eligible — outside 15-day return window"

The reasoning step shows the model **how to think**, not just what to output. This is particularly valuable for tasks with exceptions, edge cases, or compound conditions where a direct input→output pair would be ambiguous.

13.3 Structured Output Tasks: Prefix/Suffix Examples

For tasks requiring specific output formats (JSON, code, documentation), the model needs a complete example of the expected output structure. Without it, the model guesses at field names, nesting, and formatting.

Structured Output Example — JSON Format

```
{
  "order_id": "3301",
  "status": "eligible_for_return",
  "reason": "Within 30-day window",
  "next_steps": ["Initiate return label", "Schedule pickup"]
}
```

With the three example types covered, a practical question emerges: should the same examples be loaded every time, or should they vary based on the query?

13.4 Static vs. Dynamic Example Selection

Not all examples need to be present in every API call. The choice between static and dynamic selection has a significant impact on both token efficiency and output quality.

Table 11: Static vs. dynamic example selection strategies

Strategy	Description	Pros	Cons
No examples	Baseline: system prompt only	Minimal token usage	Worst performance
Static examples	Same 3 examples in every API call	Simple to implement	Wastes ~30% of tokens on irrelevant examples
Dynamic examples	Select examples matching current query via keyword matching or RAG	Relevant context; efficient token usage	Requires retrieval mechanism

Dynamic selection outperforms static because:

1. Only **relevant** examples are loaded: no wasted tokens
2. Examples match the **current query's domain** (returns query → returns examples, not shipping examples)
3. Token budget is spent on **useful** context

Dynamic selection introduces its own challenge, however: which examples should be chosen when multiple candidates exist? The answer lies in diversity.

13.5 Diversity Principle

When selecting few-shot examples, the most important criterion is not relevance alone: it is **diversity across categories**.

- **Maximize diversity** across categories
- If you have 3 slots: pick 1 return example, 1 shipping example, 1 complaint example: NOT 3 return examples
- Adding more examples beyond 3 has **diminishing returns**
- Adding non-diverse examples can actually **decrease** performance because the model overindexes on the repeated category

> ⚠ **Diversity Trap**
>
> If three examples are positive, neutral, negative, then adding two more positive examples means you now have three positive, one neutral, and one negative. The diversity suffers and the model becomes biased toward the overrepresented category.

Three diverse examples covering different categories is the sweet spot. Beyond that, additional examples cost tokens with diminishing returns. The next section shifts from prompt content to the tools that execute those prompts.

14 Coding Agents, LLMs, and Platform Configuration

A frequently confused distinction:

Table 12: LLMs vs. coding agents

Concept	Examples	What It Is
LLM (Large Language Model)	Opus, Sonnet, Haiku, GPT-4, Gemini, Qwen	The underlying model that generates text
Coding Agent	Claude Code, Codex, Cursor, Copilot, Windsurf, Antigravity	A tool built ON TOP of LLMs that provides an agentic development experience

- **Claude Code** is not an LLM: it is a coding agent built by Anthropic on top of their LLMs (Opus, Sonnet, Haiku)
- **Codex** is OpenAI's coding agent built on their LLMs
- **Google Antigravity** is Google's coding agent (launched November 2025), built on Gemini 3. It is an agent-first IDE — a VS Code fork where multiple autonomous agents work in parallel on architecture, code, tests, and UI verification simultaneously. Its configuration file equivalent is `GEMINI.md`.
- **Cursor, Copilot, Windsurf** are model-agnostic coding platforms that can use different LLMs

Table 13: Major coding agents compared

Agent	Company	Primary Model	Config File	Interface
Claude Code	Anthropic	Claude (Opus / Sonnet / Haiku)	CLAUDE.md	Terminal CLI
Codex	OpenAI	OpenAI LLMs	Markdown files	Terminal / IDE
Google Antigravity	Google	Gemini 3.1 Pro (also supports Claude, GPT)	GEMINI.md	Agent-first VS Code fork
Cursor	Anysphere	Model-agnostic	.cursor/rules	IDE (VS Code fork)
GitHub Copilot	Microsoft / GitHub	Model-agnostic	copilot-instructions.md	IDE plugin
Windsurf	Codeium	Model-agnostic	.windsurfrules	IDE (VS Code fork)
Cline	Cline (VS Code extension)	Model-agnostic	cline.md	VS Code extension

The comparison table above includes each agent's configuration file. This matters because the system prompt concepts from earlier sections (identity, rules, format, knowledge, tools) apply universally, but the **file** that encodes them is platform-specific.

> ⚠ **CLAUDE.md is Claude Code Only**
>
> CLAUDE.md is **not** a universal configuration file. It is read exclusively by **Claude Code** — Anthropic's coding agent [1]. Other coding assistants (Cursor, Copilot, Windsurf, Codex, Antigravity) do not recognise it. If you switch platforms, you must rewrite your configuration using that platform's equivalent file (e.g., `GEMINI.md` for Antigravity [5]). `AGENTS.md` is the closest thing to a universal standard [3].

15 Poorly Designed vs. Well-Designed CLAUDE.md

Poorly Designed

- No headings or subheadings (reads like plain text)
- Vague instructions ("design the architecture properly")
- No explicit technology choices
- No project structure
- Could be a .txt file: does not leverage markdown features

Well-Designed

- Clear heading hierarchy (#, ##, ###)
- Explicit technology choices ("Use Next.js, Tailwind CSS, deploy on Vercel")
- Project folder structure provided
- Specific rules with concrete guidance
- Common mistakes to avoid section (highest value per token)

The model can learn code patterns and naming conventions by reading source files — but it **cannot** learn "never use raw SQL here: always use the query builder" or "don't use floating-point for financial calculations." This **tribal knowledge** (knowledge that exists only in developers' heads and code review comments) is what the anti-pattern section encodes. No amount of code reading will surface it.

The following concrete snippets illustrate the difference:

Poorly Designed CLAUDE.md

```
This is a customer support chatbot for ShopCo. It helps
customers with returns, orders, and billing. Be helpful and
professional. Handle errors gracefully. Use the right
technologies. Don't hallucinate. Design the architecture
properly. Make sure the code is clean. The app uses React
and Node and Supabase. We deployed on Vercel last month.
Returns are usually 30 days but sometimes we make exceptions
for loyal customers. Use JSON for API responses. The team
prefers functional components. We had a bug last week where
someone used raw SQL and it broke everything, so be careful.
Tests should pass. Follow best practices.
```

</> Well-Designed CLAUDE.md

```
# Identity
Customer support agent for ShopCo. Handles return requests,
order status inquiries, and billing disputes via chat.

## Rules
- NEVER approve a return outside the 30-day window, even
  if the customer claims loyalty status
- Always consult the return policy document before making
  any return determination
- If a customer mentions a damaged item, route to the
  escalation queue instead of processing as a standard return
- Use snake_case for all variable and function names
- All API responses must include a `request_id` for tracing

## Format
- API responses: JSON with { status, reason, next_steps }
- Folder structure:
  src/
    components/    # React functional components only
    api/           # Express route handlers
    lib/           # Shared utilities
- Build: `npm run build` | Dev: `npm run dev`
- Tests: `npm run test -- --watch`

## Knowledge
- Tech stack: Next.js 14, Tailwind CSS, Supabase, Vercel
- Current phase: Phase 2 (multi-user, launched Jan 2026)
- Auth: Supabase Auth with row-level security enabled
- Return window: 30 calendar days from delivery date
- Escalation triggers: damaged items, orders above $500,
  repeat complaints (3+ in 90 days)

## Tools
- Supabase client for all database operations
- Resend for transactional email
- Sentry for error tracking (DSN in .env)

## Common Mistakes to Avoid
- Do NOT use raw SQL: always use the Supabase query builder.
  Raw SQL bypasses RLS policies and has caused data leaks.
- Do NOT use floating-point for financial calculations:
  use integer cents (e.g., 1999 = $19.99)
- Do NOT hardcode the return window (30 days): read it
  from the config table so it can be changed without deploy
- Do NOT call external APIs without a timeout: default
  to 5000ms, log and return a fallback on timeout
```

> 🔥 **Tip**
>
> If you do this, your `CLAUDE.md` file may become bigger, but it will save you a lot of pain downstream. And in any case, `CLAUDE.md` is not going to be entirely fetched all at once: selective retrieval handles the size.

16 Implementation Notes

Table 14: Practical implementation notes

Tool / Setting	Detail
Claude Code installation	Run the install command from the Claude Code landing page, then type claude in the terminal
CLAUDE.md bootstrap	Run /init in Claude Code to auto-generate a starter CLAUDE.md based on your project's build system, test framework, and code patterns
CCO (Claude Code Operator)	A thin protective layer enabling Claude Code with skip-permissions. Install from GitHub, invoke with cco instead of claude
Context visualization	Use /context in Claude Code to see token consumption breakdown by element
Terminal shortcut	Cmd + J (macOS) toggles the VS Code terminal
Markdown preview	Install Markdown Preview Enhanced VS Code extension; use Cmd + Shift + V to preview

17 Key Takeaways

- **System prompt** = `CLAUDE.md` in Claude Code. It is the single most important file you create for any project.
- `CLAUDE.md` has five components: **Identity, Rules, Format, Knowledge, Tools**. Rules are the highest priority and should be the last component removed under token constraints.
- Write at the **right altitude**: not too vague (“be helpful”), not too brittle (“reply: $9.99/month”). Target actionable guidance that remains valid as details change.
- **Start minimal, then add**: build system prompts iteratively, driven by observed failures, to avoid contradictions.

- `CLAUDE.md` is **not fully loaded every call**: sections are retrieved via keyword matching or RAG. Markdown headings create natural chunk boundaries.
- **Few-shot examples** significantly improve performance. Three diverse examples covering different categories is the sweet spot. Dynamic selection (matching examples to the query) outperforms static selection.
- **Sub-agents have their own context windows**: they make separate API calls and their context is not polluted by the parent agent's context.
- **`CLAUDE.md` is Claude Code only** — it is not read by Cursor, Copilot, Windsurf, or Codex. Each platform has its own equivalent file. **AGENTS.md** (released by OpenAI, governed by the Linux Foundation's AAIF, adopted by 60,000+ projects) is the emerging universal standard across platforms.
- **Common Mistakes to Avoid** is the highest-value-per-token section because it encodes tribal knowledge the model cannot discover by reading code.
- Modern LLMs are "incredibly good even without a good system prompt" for simple queries: the value of structured system prompts becomes clear at scale, over thousands of interactions.
- Context engineering is **not physics**: it is an evolving art based on thumb rules from community experimentation.

18 Glossary

Term	Definition
CLAUDE.md	Markdown file defining the system prompt for Claude Code projects
AGENTS.md	Open standard for defining sub-agent behavior, released by OpenAI and governed by the Linux Foundation's Agentic AI Foundation (AAIF)
SKILL.md	File defining specific capabilities or domain skills for an agent
SOUL.md	File defining the personality and character of an agent (non-Claude Code specific)
Right Altitude	Anthropic's principle that instructions should be neither too vague nor too specific
Lost in the Middle	Attention degradation for content placed in the center of a long context window
Context Rot	Gradual degradation of model performance as the context window fills with stale information

Few-Shot Examples	Sample input/output pairs provided in the prompt to demonstrate desired behavior
Static Examples	Fixed set of examples loaded in every API call regardless of the query
Dynamic Examples	Examples selected at inference time based on relevance to the current query
CCO	Claude Code Operator: a tool that wraps Claude Code with skip-permissions for smoother long builds
Wispr Flow	Voice-to-text macOS tool for rapidly capturing requirements
Tribal Knowledge	Knowledge that exists only in developers' heads and code reviews, not in the codebase itself
Chain-of-Thought (CoT)	Few-shot pattern showing input → reasoning → output to teach multi-step decision-making
Prefix/Suffix Pattern	Few-shot pattern showing desired output structure/format via complete examples

19 Notation Reference

Table 15: Notation reference

Symbol / Abbreviation	Meaning
CLAUDE.md	System prompt file for Claude Code
#, ##, ###	Markdown heading levels (H1, H2, H3)
RAG	Retrieval-Augmented Generation
MCP	Model Context Protocol
LLM	Large Language Model
MVP	Minimum Viable Product
API	Application Programming Interface

20 Open Questions / Areas for Further Study

1. **Production system prompt portability:** When moving from Claude Code (`CLAUDE.md`) to other APIs (OpenAI, Gemini), how much of the system prompt content can be reused, and what must be rewritten for each platform's conventions?
2. **Context rot mitigation:** Beyond reinjecting system prompt rules, what are systematic strategies for maintaining performance in long-running agent sessions?
3. **XML vs. Markdown empirical comparison:** Are there rigorous A/B tests showing output quality differences between XML-structured and Markdown-structured system prompts across different models?
4. **Optimal `CLAUDE.md` size:** Is there a token count beyond which `CLAUDE.md` quality degrades even with selective retrieval?
5. **Dynamic few-shot selection algorithms:** Beyond keyword matching, what retrieval strategies (semantic search, hybrid) produce the best dynamic example selection?
6. **`SOUL.md` impact:** Rigorous evaluation of the claimed "night and day" difference when using `SOUL.md` : what specific aspects of output quality change?

References

[1] Anthropic, "Best Practices for Claude Code." [Online]. Available: https://code.claude.com/docs/en/best-practices

[2] Anthropic, "How Claude Remembers Your Project — CLAUDE.md Documentation." [Online]. Available: https://code.claude.com/docs/en/memory

[3] OpenAI, "AGENTS.md: Open Standard for Agent Configuration." [Online]. Available: https://agents.md/

[4] Linux Foundation, "Agentic AI Foundation (AAIF)." [Online]. Available: https://aaif.io/

[5] Google, "Google Antigravity Documentation — Rules and Workflows." [Online]. Available: https://antigravity.google/docs/rules-workflows

Follow me for More AI Content

If you found these notes useful, connect with me on LinkedIn for more deep dives into Machine Learning, Artificial Intelligence, and Computer Vision.

Isham Rashik on LinkedIn

Scan the QR code or click the link above