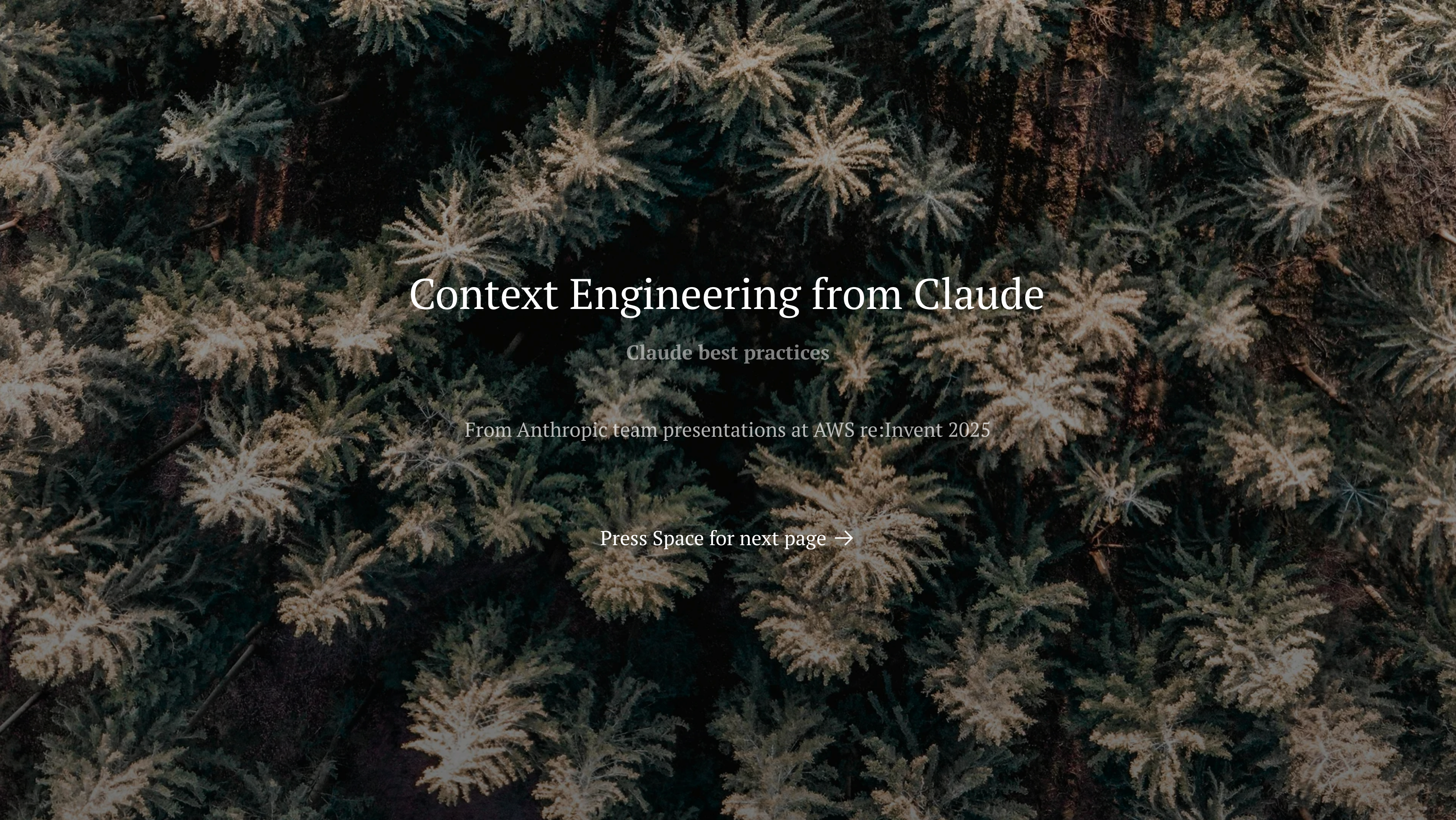


Context Engineering from Claude

Claude best practices

From Anthropic team presentations at AWS re:Invent 2025

Press Space for next page →

Agenda

Core Problem: Why Context Engineering?

Claude is smart — intelligence is not the bottleneck. Context is.

Organizations have unique workflows, procedures, and institutional knowledge that Claude does not know about.

Claude does not know:

- How your team structures reports
- Your brand guidelines and templates
- Your compliance procedures
- Your data analysis methodologies

Current solutions fall short:

- Prompts are ephemeral instructions
- Custom agents require building infrastructure
- Context management is challenging

Part I: Skills

What are Skills?

Skills are organized **folders of instructions, scripts, and resources** that Claude can discover and load dynamically. Think of them as "expertise packages."

Type 01: General capabilities

Claude is not good at out of the box (yet)

e.g. creating PDFs, Excel, & PowerPoint files

Type 02: Workflows & best practices

An org’s / vertical’s / individual’s workflows

e.g. Anthropic's brand styling

How Skills work — a simple directory

Skill is a directory containing SKILL.md file.

- **Metadata:** File starts with name and description
- **Preloaded:** Agent pre-loads name/description into system prompt
- **Efficient:** Claude only reads more when needed
- **Discovery:** Claude navigates and discovers detail as needed
- **Executable Scripts:** Token efficient, deterministic reliability

```
## pdf/SKILL.md

name: pdf
description: PDF toolkit for extracting
text/tables, merging/splitting, forms

## Overview
PDF processing with Python libraries.
For advanced: `/reference.md`
For forms: `/form.md`

## Quick Start
from pypdf import PdfReader
reader = PdfReader('document.pdf')
```

Progressive disclosure

anthropic/brand_styling/SKILL.md

```
name: Anthropic Brand Style Guidelines
description: Brand identity resources for presentations
```

```
## Colors
Dark: '#141413' - Primary text
Light: '#faf9f5' - Light backgrounds
Light Gray: '#c8c6dc' - Subtle backgrounds

## Workflows
Presentations → `./slide-decks.md`
Documents → `./docs.md`
```

anthropic/brand_styling/slide-decks.md

```
## Anthropic Slide Decks
Intro/outro: bg '#141413', fg oat
Section: bg '#da7857', fg '#141413'
```

anthropic/brand_styling/docs.md

```
## Documents
Start with title, authors, creation date
If using GDocs tabs, title main doc accordingly
```

Claude reads only what's needed: slide-decks.md for presentations, docs.md for documents

Skills work in all our products

Apps

- **Best for:** automatically invoked, user experience
- **Foundational Skills:** professional documents and analysis
- **Custom Skills:** users create, manage, share

Developer Platform

- **Best for:** programmatic distribution
- **Deploy Skills:** via Code Execution API
- **Foundational or Custom:** core or custom skills

Claude Code

- **Best for:** developer workflows
- **Auto-invoked:** Claude loads automatically (vs slash commands)
- Runs in local dev environment
- Install via Plugins or `~/.claude/skills`
- **Marketplace:** distributed via plugin marketplace

Skills best practices

Naming and descriptions

• Use gerund form: `processing-pdfs`

• Avoid vague names (`helper`, `utils`)

• Include what it does AND when to use

• Be declarative: "Processes Excel files"

• Avoid: "I can help you..." or "You can use this to..."

File organization

• Keep SKILL.md under 500 lines

• Split when approaching limit

• Keep references one level deep

• Structure longer files (>100 lines) with TOC

Content quality

• Use consistent terminology

• Show concrete input/output pairs

• Examples align with desired behaviors

Skills examples

01
Code Security Agent

02
Code Review Agent

03
Contract Review Agent

04
Meeting Summary Agent

05
Financial Reporting Agent

06
Email Automation Agent

07
Invoice Processing Agent

Built with Claude Agent SDK

Part II: Context Engineering Framework

Context Engineering — Four Pillars

The discipline of optimizing the utility of tokens against the inherent constraints of LLMs

System prompt

- Minimal, precise instructions
- "Say less, mean more"
- Structured sections
- Right altitude (not too rigid, not too vague)

Tools

- Self-contained, no overlap
- "Every tool earns its place"
- Explicit parameters & concise descriptions
- Clear success/failure patterns

Data retrieval

- JIT Context
- "Load what you need, when you need it"
- Balance pre-loading vs dynamic fetching
- Don't send the library. Send a librarian.

Long horizon

- Compaction strategy
- Structured note-taking
- Sub-agent architecture

Data retrieval paradigm shift

Old approach: Pre-Loading (Traditional RAG) — Load all potentially relevant data upfront
New approach: Just-In-Time

Lightweight identifiers

- Pass IDs, not full objects
- Agent requests details if needed
- **Example:** `user_id: "12345"` → agent calls `get_user()` → Full profile

Progressive disclosure

- Start with summaries
- Agent drills down as needed
- **Example:** File list → File metadata → File contents

Autonomous exploration

- **Agentic Search:** Give discovery tools, not data dumps
- Agent navigates information space
- **Example:** `search_docs()` + `read_doc(level)` vs loading all

Three strategies for long-horizon tasks

When tasks exceed context window capacity

Compaction

- Periodically summarize and compress history
- Reset context with compressed summary
- Trade: Minor detail loss for continued operation
- **Example:** "User wants X, tried Y, learned Z"

Structured memory

- Explicit memory artifacts (external storage)
- Store decisions, learnings, state
- Retrieved on-demand
- **Example:** Decision log, key findings doc

Sub-Agent architectures

- Decompose into specialized agents
- Each has focused, narrow context
- Main agent orchestrates
- **Example:** Code-review spawns doc-checker

Part III: Context Window & Context Rot

Context Window & Context Rot

Context window

All frontier models have a maximum number of total tokens able to be processed in a single exchange.

Anthropic’s context window is **200k tokens**.

Context rot

As context grows, output quality can regress.

Reasons:

01. Poisoning — incorrect/outdated info

02. Distraction — irrelevant info

03. Confusion — similar info mixed

04. Clash — contradictory info

Chroma Report: Context-Rot: How Increasing Input Tokens Impacts LLM Performance

Prompt caching & Benefits

Prompt caching is a lever for **cost & latency**

Prompt caching success is highly correlated with **structure of context**

Effectively building and maintaining context will:

- Handle context window limits → **Reliability**
- Reduce context rot → **Accuracy**
- Optimize for prompt caching → **Cost & latency**

Part IV: Tool Design Best Practices

Elements of strong tool design

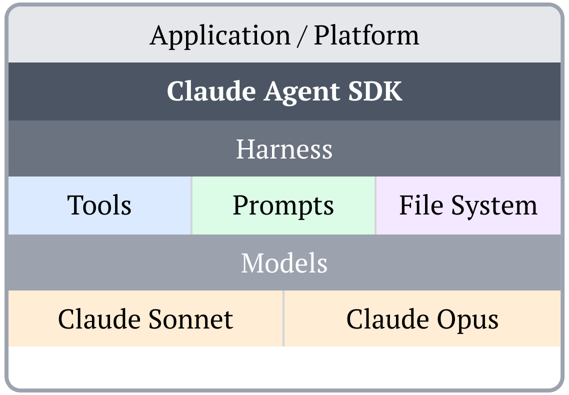
- **Simple & accurate tool name**
- **Detailed descriptions** — Include what the tool returns, how it should be used
- **Avoid similar names/descriptions**
- **Single action per tool** — at most 1 level of nested parameters
- **Provide examples** — expected input/output format
- **Pay attention to results format**
- **Test your tools** — make sure agents use them well

```
{
  "name": "search_customers",
  "description": "Search customer database by
  name, email, or ID.",
  "input_schema": {
    "type": "object",
    "properties": {
      "query": {
        "type": "string",
        "description": "Search term"
      },
      "max_results": {
        "type": "integer",
        "default": 10
      }
    },
    "required": ["query"]
  }
}
```

Part V: Claude Agent SDK

Claude Agent SDK architecture

Built on the agent harness that powers Claude Code, providing all building blocks for production-ready agents.



Core capabilities:

- **Tools:** Read/write files, code execution, web search, MCP, Skills
- **Permissions:** Human confirmation, fine-grained, allow/deny lists
- **Production:** Session management, error handling, monitoring

Enhancements:

- Subagents, Web Search, Research Mode
- Auto Compacting, Multi Stream, Memory

SDK philosophy

Claude Code — Delegate everyday dev work

By giving Claude access to the user's computer (via terminal), it can **write code like a programmer**.

- Find files, Write & edit files
- Test & Debug
- Take actions iteratively

Claude Agent SDK — Extend to custom agents

The Claude Code principle can be extended to agents in general.

- Read CSV files, Search the web
- Build visualizations, etc.

Key Design Principle: Claude Agent SDK gives your agents a computer, allowing them to work like humans do

Claude Code Toolkit

Tool	Description	Permission
Agent	Runs a sub-agent for complex, multi-step tasks	No
Bash	Executes shell commands	Yes
Edit / MultiEdit	Targeted edits to files (atomic)	Yes
Glob / Grep / LS	Find files, search patterns, list directories	No
NotebookEdit / NotebookRead	Jupyter notebook operations	Yes / No
Read / Write	Read and write files	No / Yes
TodoRead / TodoWrite	Task list management	No
WebFetch / WebSearch	Fetch URL content, web search with domain filtering	Yes

Best agentic frameworks

- Do not overly scaffold the models
- Allow for tuning all key parts of the system (Context Engineering)
- Leverage all model capabilities (Extended & Interleaved Thinking, Parallel Tool Calling, etc)
- Provide access to memory
- Enable multi-agents, where valuable
- Have robust agent permissioning

Part VI: Subagent Configuration

Subagent configuration best practices

Description field

• Critical for auto-invocation

• Make specific and action-oriented

• Use "PROACTIVELY" or "MUST BE USED"

• e.g. "Use PROACTIVELY when code changes might impact performance"

Tool permissions

• Limit tools to what each subagent needs

• Example: code-reviewer gets `Read, Grep, Glob` but not `Write` or `Edit`

Model selection

• Use `inherit` to match main conversation

• Specify `sonnet`, `opus`, or `haiku`

• Default is `sonnet` if omitted

Native subagent orchestration

Managing context limits

• When context window clears, consider starting fresh rather than compacting

• Prompt around early compaction

• Be prescriptive about how it should start

• Provide verification tools

• Claude needs to verify correctness without continuous human feedback

For optimal research results

• Provide clear success criteria

• Encourage source verification across multiple sources

• Use a structured approach for complex research

Part VII: MCP (Model Context Protocol)

What is MCP?

Chat interface
Claude Desktop, LibreChat

IDEs and code editors
Claude Code, Goose

Other AI applications
Sire, Superinterface

MCP
Standardized protocol

↔ Bidirectional data flow ↔

Data and file systems
PostgreSQL, SQLite, GDrive

Development tools
Git, Sentry, etc.

Productivity tools
Slack, Google Maps, etc.

Where is MCP heading?

Last spec (June 2025) focused on structured tool outputs, OAuth authorization, elicitation for server-initiated interactions, and security best practices

1. **Asynchronous operations**

2. **Statelessness and scalability**

3. **Server identity and discovery**

Part VIII: Evaluations

Evaluations — Tips

Good evals:

- Measure performance and regressions
- User-centric, cover full range of expected behaviors
- Consider edge cases and risks
- Have buy-in from multiple stakeholders
- Have a target
- Use existing benchmarks if available

Important:

- Negative examples are extremely important — they define the boundary and prevent over-triggering
- The "harder" negative examples = higher quality feature
- Brainstorm (with others or Claude) how the model might over-apply your feature

Great evals:

Can be graded objectively and programatically

Evaluations — Types

Eval type	Description	Used for	Examples
Intelligence Benchmarks	General intelligence	Model comparison, launch decisions	MMLU, GPQA
Capability Benchmarks	Specific capability (coding, math)	Model positioning	MATH, HumanEval, SWE-Bench, Tau-bench
Behavioral Evals	Particular model behaviors	Monitor/improve behaviors	Refusals, Hallucinations, "Certainly!"
Safety evals	Threat analysis, red teaming	Safety perspective	Redteaming Computer use
Product Evals	Task in product context	Product decisions	Artifacts, Multimodal PDFs

Evaluations — Grading & Iteration

Grading process:

1. Determine Baseline — Run prompts with current production config, record output

2. Outline Expected Behavior — For positive and negative examples

3. [Optional] Grading — Build a grader (exact match, regex, or model-based). Run on baseline: negatives score well, positives don't (until your change)

Automate and iterate:

- The faster you can run your eval, the more easily you can iterate
- **Tooling:** Anthropic Console, custom script/notebook, custom tools
- **Feature iteration:** change system prompt, tool definitions
- **Eval iteration:** add use-cases for unexpected behaviors

Examples of evals for agents

Answer accuracy — LLM judges correctness

- **User:** How many employees started in 2023?
- **Agent:** query_employees(hired_year=2023, status="active")
- **Tools:** {"count": 47, ...}
- **Agent:** 47 employees. Engineering (23), Sales (15), Ops (9).
- **LLM Judge:** CORRECT — Score: 10/10

Tool use accuracy — Correct tool selection

- **User:** Book a flight to Paris tomorrow
- **Agent:** search_flights(date="tomorrow")
- **Tools:** {"error": "Invalid date format"}
- **Agent:** search_flights(departure_date="2024-05-19")
- **Pass: Recovered from error**

t-bench — Agent reaches correct final state (e.g., flight cancellation flow with user interruption)

Tips for evaluating agentic systems

- **The larger the effect size, the smaller the sample size needed:** Starting out, you just need a few test cases. Each change will have substantial, noticeable impact.
- **Use realistic tasks:** Evaluate on tasks real users might use, with clear correct answers findable using available tools.
- **LLM-as-judge with a rubric is very powerful:** LLMs are strong judges if given a clear rubric aligned with human judgements.
- **Nothing replaces human evals:** Bashing + vibe checking, and testing with real users — humans find the rough edges!

Part IX: Building Coding Agents

What we learned about building coding agents

Key insights:

- Everything is a File
- Bash is the ultimate tool
- Most toolcalls are just code
- Agentic Search > RAG

Agents also need:

- Memory
- Sub Agents & Collaboration
- Dynamic Tool Calls
- Code Generation & Execution
- Web Search / Agentic Search
- Long Running Tasks

Part X: Ecosystem Collaboration

The ecosystem — How they work together

Feature	Prompts	MCP	Skills	Subagents
Provides	Instructions	Tool connectivity	Procedural knowledge	Task delegation
Persistence	Single conversation	Continuous	Across conversations	Across sessions
Contains	Natural language	Tool definitions	Instructions + code	Full agent logic
Can contain code	No	Yes	Yes	Yes
When loads	Each turn	Always available	Dynamically	When invoked
Best for	Quick requests	Data access	Specialized expertise	Specialized tasks

Example workflow & Matching tools

Example agentic workflow:

1. MCP connects to Google Drive and GitHub
2. Skills provide analytical framework (competitive analysis)
3. Subagents execute in parallel (market-researcher, technical-analyst)
4. Prompts refine and provide specific context

Matching the right tool to use case:

- Procedural knowledge needed repeatedly → **Skill**
- Access to external data sources → **MCP**
- Independent execution with separate context → **Subagent**
- Complex workflow → **combine all three**

Context Engineering from Claude

Claude is smart enough — the key to success is giving it the right context.

From Anthropic team presentations at AWS re:Invent 2025

Powered by Slidev