

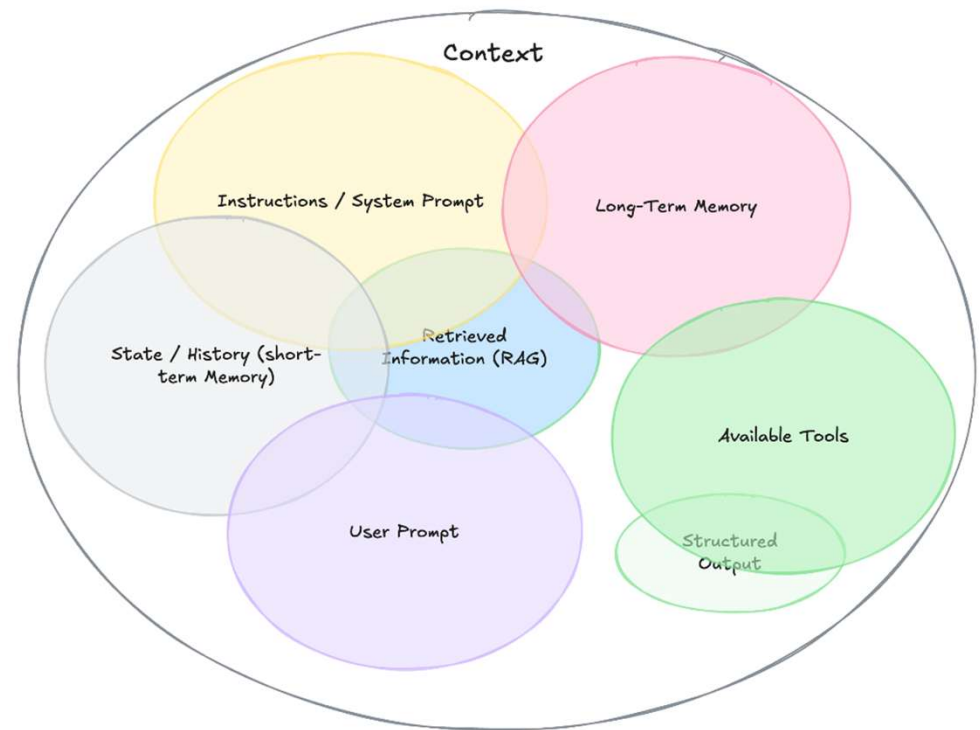
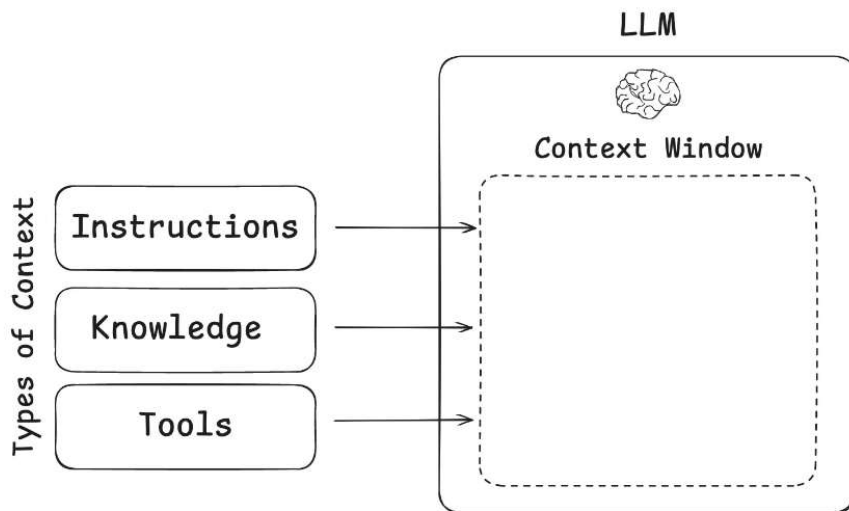
Context Engineering

IE685 Large Language Models and Agents



Definition: Context Engineering

Context engineering is the delicate art and science of filling the context window with just the right information for the next step." — Andrej Karpathy



Outline

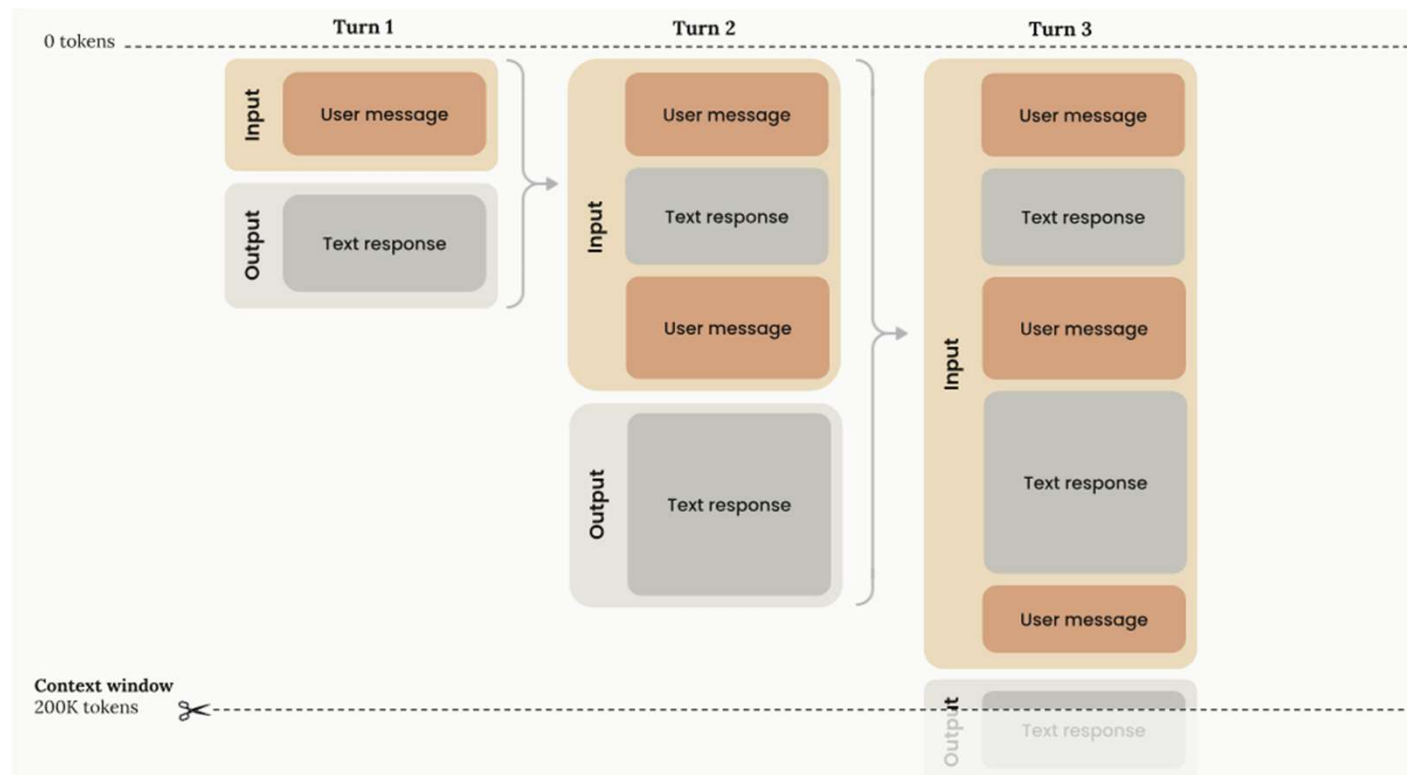
1. Motivation for Context Engineering
 1. Context Rot
 2. Token Costs
 3. Prompt Caching
2. Context Engineering as Architectural Thinking
3. Context Engineering Techniques
 1. Write Context
 2. Select Context
 3. Compress Context
 4. Isolate Context

1. Motivation for Context Engineering



The Context Window Fills Quickly

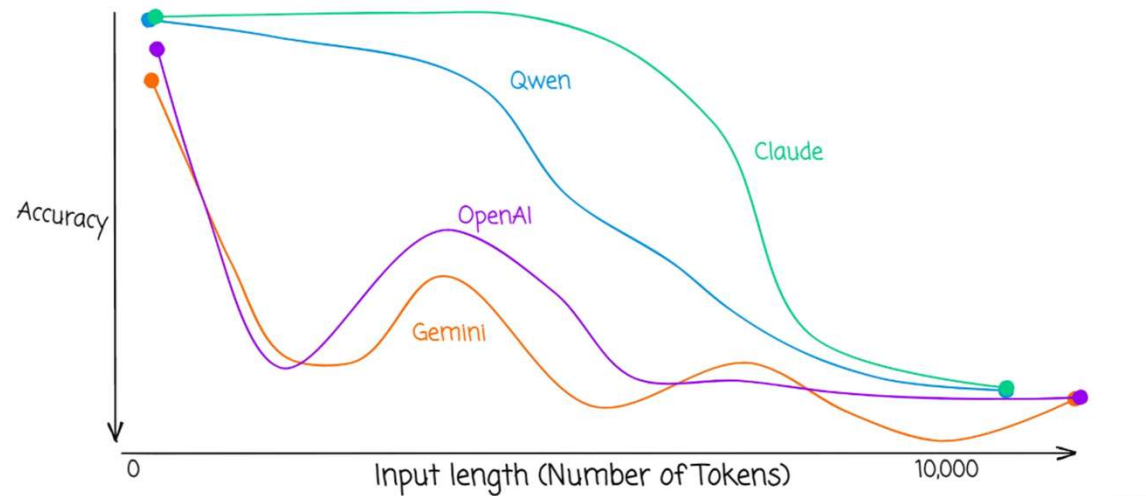
- current models have context windows of 128k to 1M tokens
 - 128k tokens = ~200 pages of text
- but repeated tool calls and longer running tasks quickly fill these context windows



Context Rot

Degradation of a LLM’s accuracy and reliability as the length of its input context increases

Context Rot: How Increasing Input Tokens Impacts LLM Performance



Source: <https://research.trychroma.com/context-rot>

COBUS GREYLING & AI

Needle in the haystack problem



<https://research.trychroma.com/context-rot>

Costs of LLM Calls

- cost of a single API call with 128k tokens
 - GPT5.2: \$0.22/call, GPT5.2 mini: \$0.03/call
- per task costs of Web agents performing product search (6 – 20 calls/task)

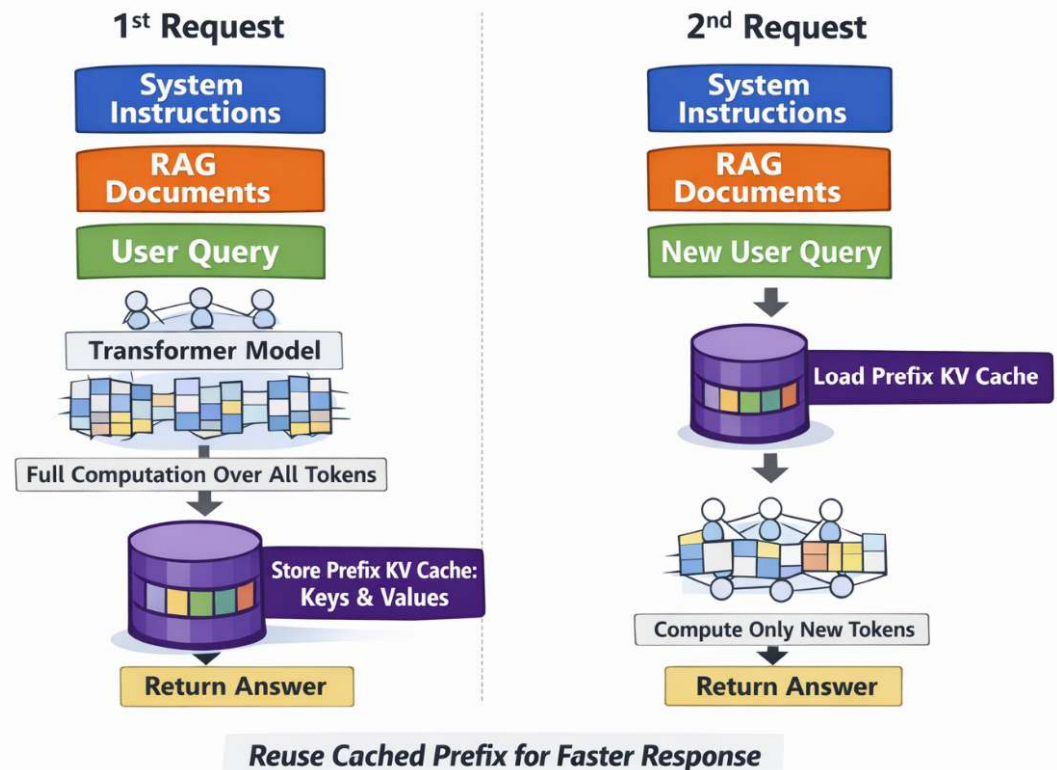
Agent	CR	F1	Token	Cost	Runtime
HTML	0.57	0.67	241,136	\$0.52	291 s
RAG	0.68	0.77	46,667	\$0.10	50 s
MCP	0.62	0.75	139,569	<u>\$0.27</u>	62 s
NLWeb	<u>0.64</u>	<u>0.76</u>	<u>71,214</u>	\$0.10	<u>53 s</u>

➔ agents are expensive and slow!

Steiner et al.: MCP vs RAG vs NLWeb vs HTML: A Comparison of the Effectiveness and Efficiency of Different Agent Interfaces to the Web. The World Wide Web Conference, 2026.

Prompt Caching

- stores the state of the Transformer (keys and values) after processing a part of the prompt (prefix) in a cache
- if second query has same prefix, the state is loaded and not re-computed
 - saves compute
 - but keys and values for 100K tokens require several GB of memory
 - stored in GPU memory or on same machine
 - next request is routed to the same machine based on prefix



Discounts for Cached Tokens

Model	Input (per 1M tokens)	Cached input (per 1M tokens)	Caching Discount
gpt-4o	\$2.50	\$1.25	50.00%
gpt-4.1	\$2.00	\$0.50	75.00%
gpt-5-nano	\$0.05	\$0.005	90.00%
gpt-5.2	\$1.75	\$0.175	90.00%
gpt-realtime (audio)	\$32.00	\$0.40	98.75%

Model	Base Input Tokens	5m Cache Writes	1h Cache Writes	Cache Hits & Refreshes	Output Tokens
Claude Opus 4.6	\$5 / MTok	\$6.25 / MTok	\$10 / MTok	\$0.50 / MTok	\$25 / MTok
Claude Opus 4.5	\$5 / MTok	\$6.25 / MTok	\$10 / MTok	\$0.50 / MTok	\$25 / MTok
Claude Opus 4.1	\$15 / MTok	\$18.75 / MTok	\$30 / MTok	\$1.50 / MTok	\$75 / MTok
Claude Opus 4	\$15 / MTok	\$18.75 / MTok	\$30 / MTok	\$1.50 / MTok	\$75 / MTok
Claude Sonnet 4.6	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok
Claude Sonnet 4.5	\$3 / MTok	\$3.75 / MTok	\$6 / MTok	\$0.30 / MTok	\$15 / MTok

<https://developers.openai.com/api/docs/guides/prompt-caching/>
<https://platform.claude.com/docs/en/build-with-claude/prompt-caching/>

Caching Speeds up Answers

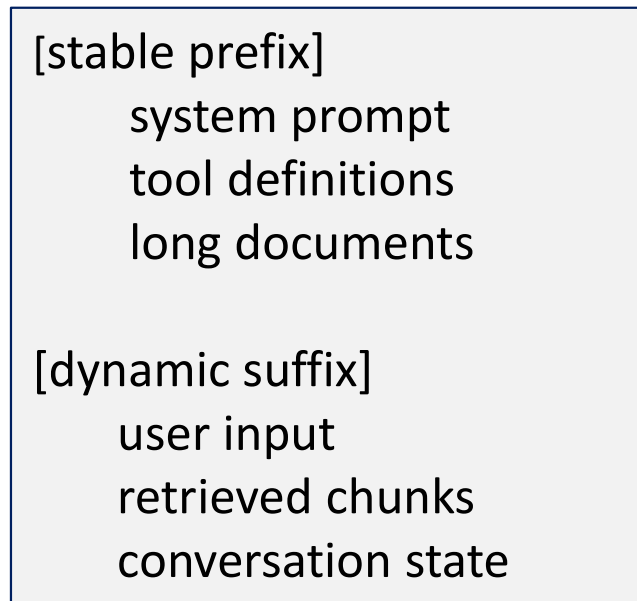
TTFT = time to first token



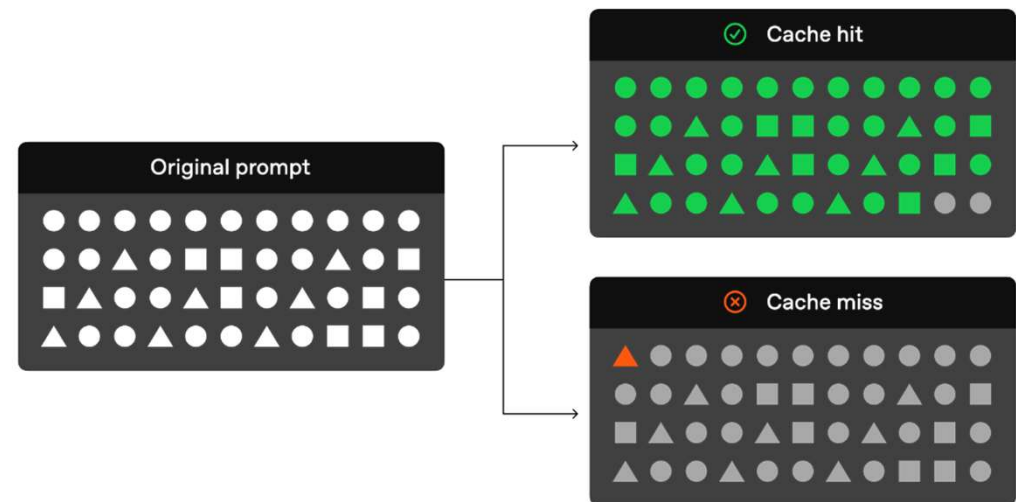
https://developers.openai.com/cookbook/examples/prompt_caching_201/

Implication for Context Engineering

- **Put stable content blocks first!**

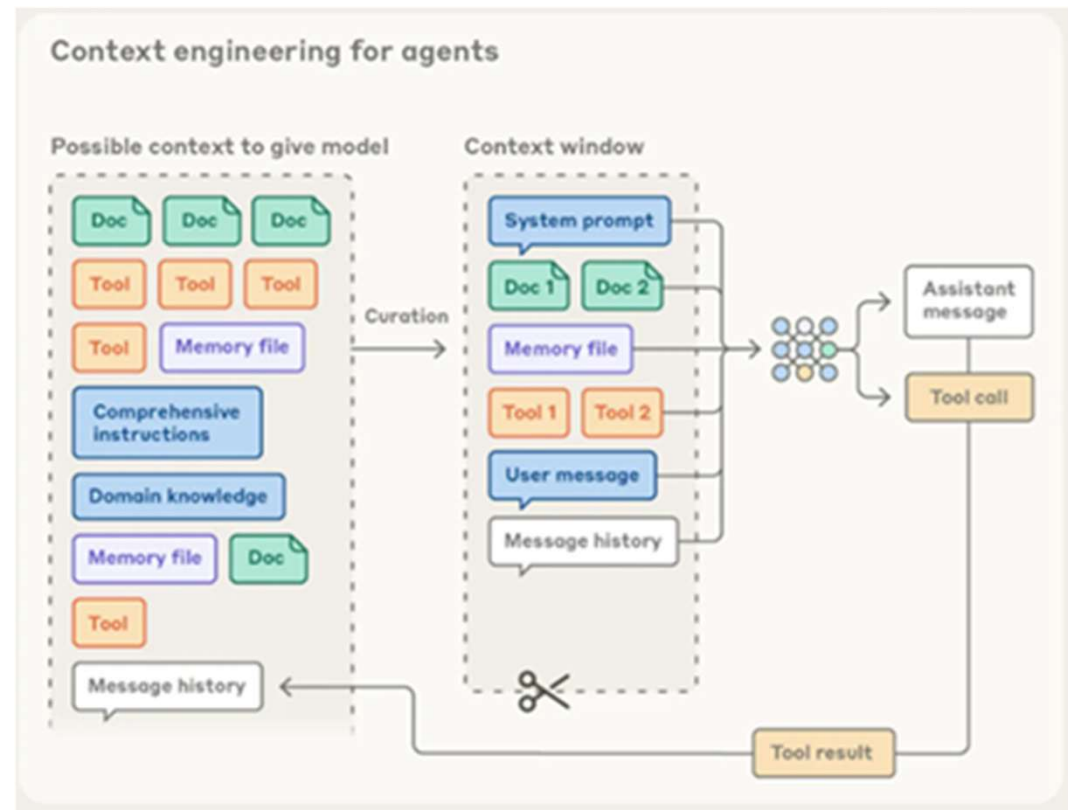


- **Avoid cache misses and updates as they are expensive!**



2. Context Engineering as Architectural Thinking

- Think in **content blocks!**
- Which blocks are relevant for the LLM **right now?**
- What is the **best order** of these blocks to
 - get good answers and
 - avoid cache mixes?

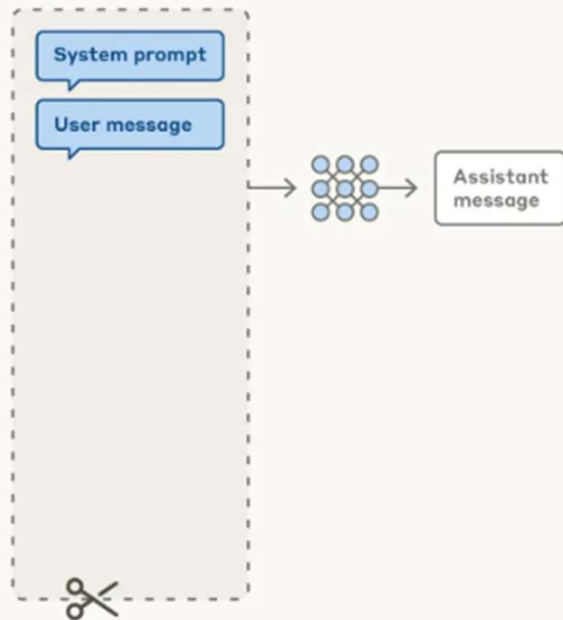


Prompt vs. Context Engineering

Prompt engineering vs. context engineering

Prompt engineering for single turn queries

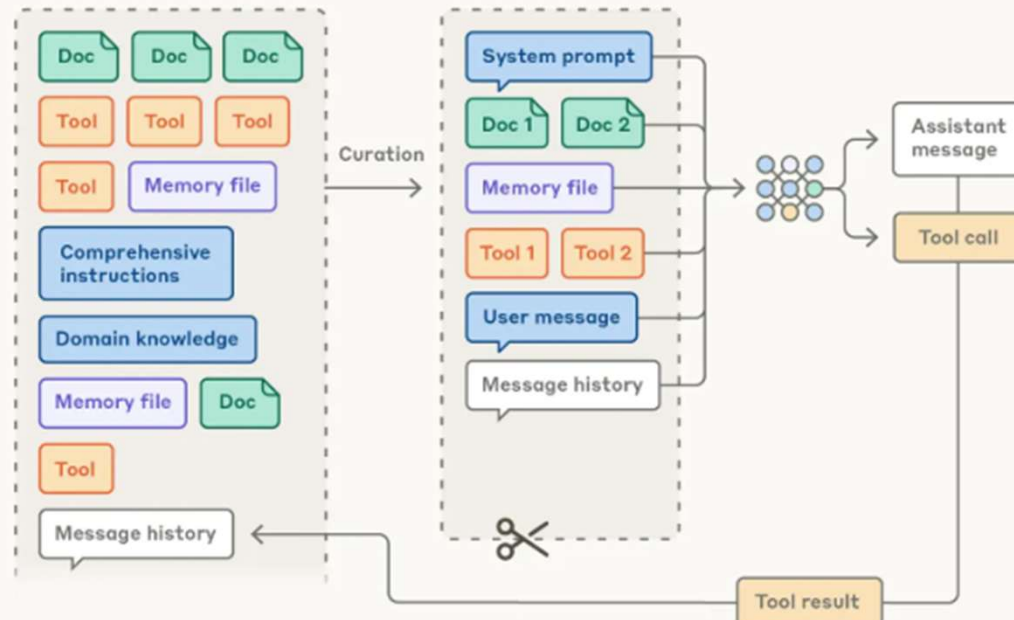
Context window



Context engineering for agents

Possible context to give model

Context window



<https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents/>

Example: Claude Code Context

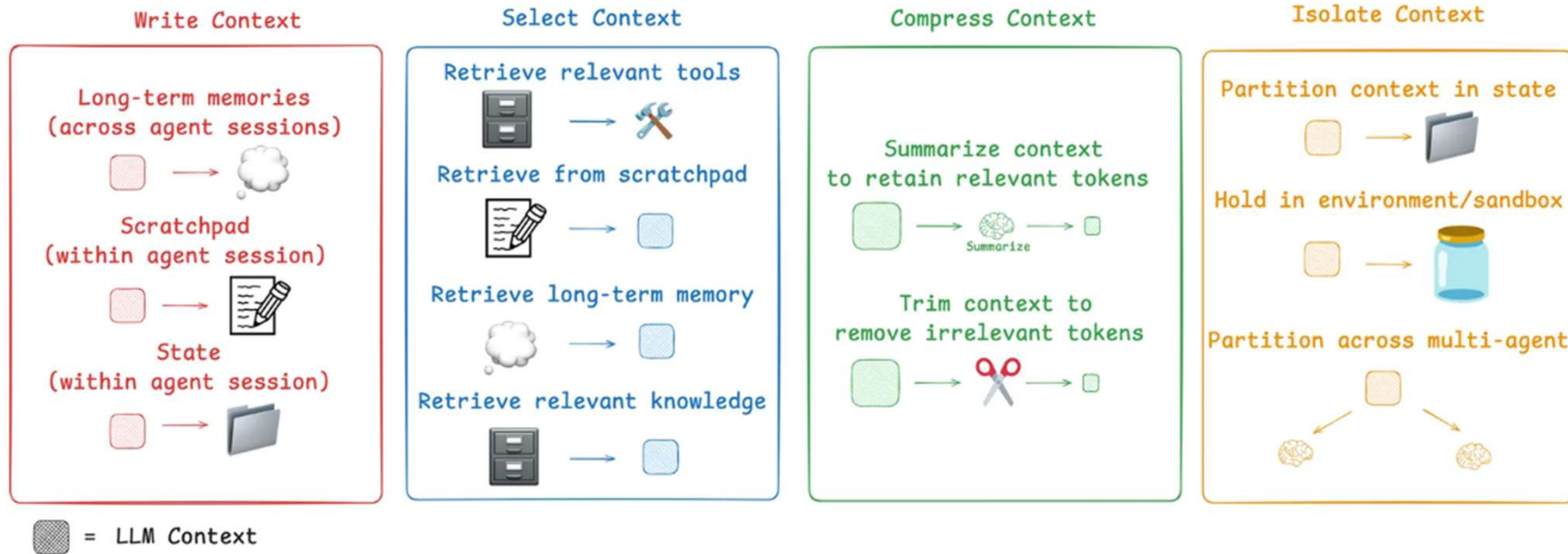
```
> /context
└─ Context Usage
   ┌─ claude-opus-4-5-20251101 · 124k/200k tokens (62%)
   │
   │   Estimated usage by category
   │   ┌─ System prompt: 2.7k tokens (1.3%)
   │   │
   │   │   System tools: 16.4k tokens (8.2%)
   │   │
   │   │   MCP tools: 293 tokens (0.1%)
   │   │
   │   │   Memory files: 23 tokens (0.0%)
   │   │
   │   │   Skills: 267 tokens (0.1%)
   │   │
   │   │   Messages: 105.2k tokens (52.6%)
   │   │
   │   │   Free space: 42k (21.0%)
   │   │
   │   │   Autocompact buffer: 33.0k tokens (16.5%)
   │
   └─ MCP tools · /mcp
      ┌─ mcp__ide__getDiagnostics: 111 tokens
      └─ mcp__ide__executeCode: 182 tokens

Memory files · /memory
└─ ~/.claude/CLAUDE.md: 23 tokens

Skills · /skills

Plugin
┌─ dev-browser: 97 tokens
┌─ frontend-design: 67 tokens
┌─ ralph-loop:help: 16 tokens
┌─ ralph-loop:ralph-loop: 14 tokens
└─ ralph-loop:cancel-ralph: 12 tokens
```

3. Context Engineering Techniques



https://rlancemartin.github.io/2025/06/23/context_engineering/
<https://www.blog.langchain.com/context-engineering-for-agents/>

Context Engineering Techniques

- **Writing context:** explicitly write down context (outside the context window) to support an agent in performing a task
- **Selecting context:** pull relevant information into the context window as needed to help an agent perform a task and keep the context focused
- **Compressing context:** slimming down large contexts by retaining only relevant parts required to perform a task
- **Isolating context:** split up context (e.g. among subagents) or isolate large context to provide only the relevant context for a specific (sub-)task

Interplay of System Prompt and Agent Harness

The techniques are implemented using a combination of

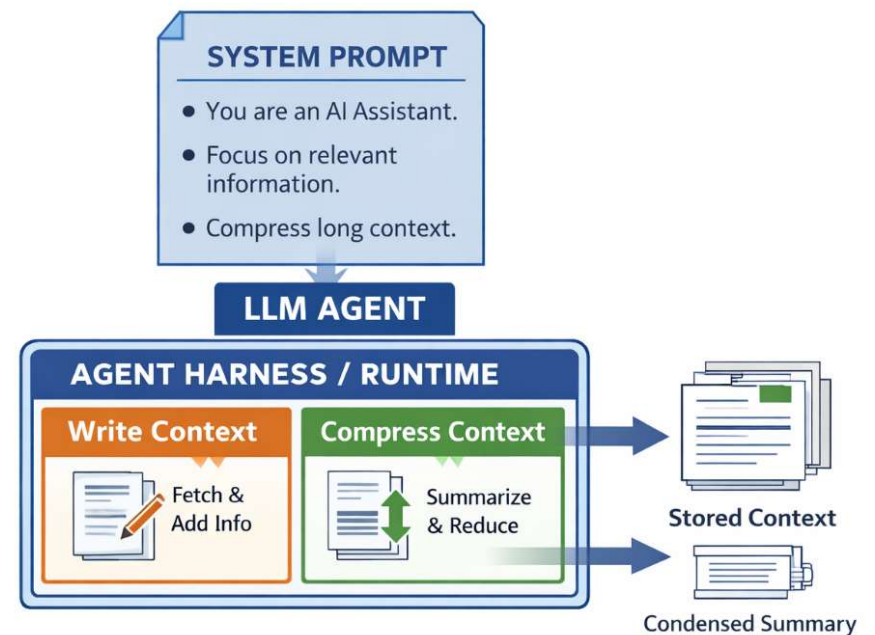
1. Instructions in the system prompt

- explain the agent which features/tools are available
- explain how it should use them

2. functionality implemented in the agent harness

- store information in files or key/value store
- retrieve information, tool descriptions, demonstrations
- call LLM to compress history
- manage communication between multiple agents

Context Engineering for LLM Agents



3.1 Write Context

- Store information that might be relevant later **outside** the context window
 - just keep information that is **relevant for the next LLM call**
 - helps LLM focus on the task at hand
 - allows on-demand retrieval of stored information
 - remember experiences from solving previous tasks
 - store intermediate results (half written paper in RAG scenario)
- Common mechanisms
 - **Memory**
 - short-term: state object that persists session
 - long-term: file, vector store, or key/value store
 - **Scratchpads**

Scratchpad

- allows the agent to take temporary notes while working on solving a task, similar to what humans do
- Can free up context window by preserving e.g.
 - plans
 - intermediate findings
 - other relevant content
- Can be implemented as
 - a tool that writes to a file
 - a field in persistent runtime/session state (in-memory)
- Stores information only for current task/sessions

Example: Scratchpad in LangChain

Enable scratchpad during agent creation:

```
from langchain.agents import create_agent
from langchain.agents.middleware import TodoListMiddleware

agent = create_agent(
    model="gpt-4.1",
    tools=[read_file, write_file, run_tests],
    middleware=[TodoListMiddleware()],
)
```

Addition to system prompt:

`write_todos`

You have access to the `write_todos` tool to help you manage and plan complex objectives. Use this tool for complex objectives to ensure that you are tracking each necessary step and giving the user visibility into your progress.

This tool is very helpful for planning complex objectives, and for breaking down these larger complex objectives into smaller steps. It is critical that you mark todos as completed as soon as you are done with a step.

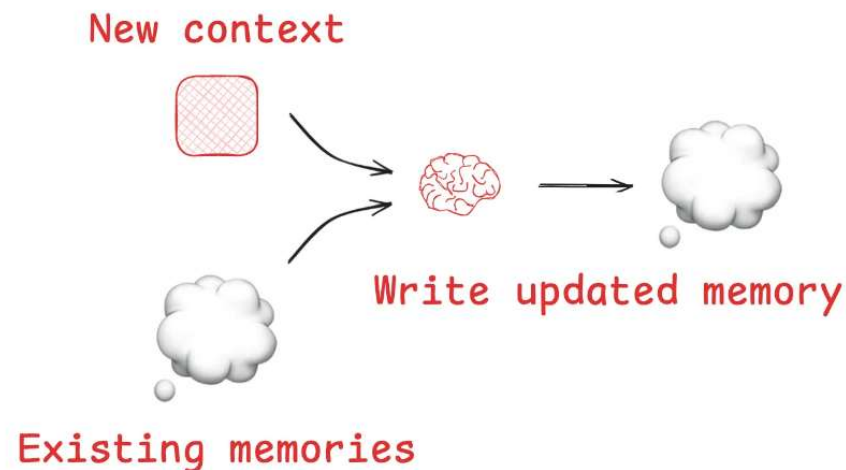
Important To-Do List Usage Notes to Remember

- The `write_todos` tool should never be called multiple times in parallel.
- Don't be afraid to revise the To-Do list as you go. New information may reveal new tasks that need to be done, or old tasks that are irrelevant.

<https://docs.langchain.com/oss/python/langchain/middleware/built-in#to-do-list>

Long Term Memory

- Extend persistence **beyond single task/session**
- Agent can retain preferences, useful prior interactions, ...
- Widely adopted in tools like ChatGPT, Claude, Cursor, ...
- Realized via saving to file or using a vector store
- LLMs can create and update memories automatically



Long Term Memory

- Different types of information to be remembered

Memory Type	What is Stored	Human Example	Agent Example
Semantic	Facts	Things I learned in school	Facts about a user
Episodic	Experiences	Things I did	Past agent actions
Procedural	Instructions	Instincts or motor skills	Agent system prompt

- Memory can be generated by e.g.
 - **Self-Reflection** after each “turn” in an agentic workflow
 - **Periodic synthesis** from past feedback and interactions

Memory in LangChain

- enable long-term memory in Langchain

```
from langchain.agents import create_agent
from deepagents.middleware import FilesystemMiddleware
from deepagents.backends import CompositeBackend, StateBackend, StoreBackend
from langgraph.store.memory import InMemoryStore

store = InMemoryStore()

agent = create_agent(
    model="claude-sonnet-4-6",
    store=store,
    middleware=[
        FilesystemMiddleware(
            backend=lambda rt: CompositeBackend(
                default=StateBackend(rt),
                routes={"/memories/": StoreBackend(rt)}
            ),
            custom_tool_descriptions={
                "ls": "Use the ls tool when...",
                "read_file": "Use the read_file tool to..."
            } # Optional: Custom descriptions for filesystem tools
        ),
    ],
)
```

Memory in LangChain

Addition in system prompt (abbreviated):

```
<agent_memory>
```

```
/AGENTS.md
```

```
# Project Context ...
```

```
</agent_memory>
```

<memory_guidelines>

The above <agent_memory> was loaded in from files in your filesystem. As you learn from your interactions with the user, you can save new knowledge by calling the `edit_file` tool.

****When to update memories:****

- When the user explicitly asks you to remember something (e.g., "remember my email", "save this preference") ...

****When to NOT update memories:****

- When the information is temporary or transient (e.g., "I'm running late", "I'm on my phone right now") ...

****Examples:****

Example 1 (remembering user information):

User: Can you connect to my google account?

Agent: Sure, I'll connect to your google account, what's your google account email?

User: john@example.com

Agent: Let me save this to my memory.

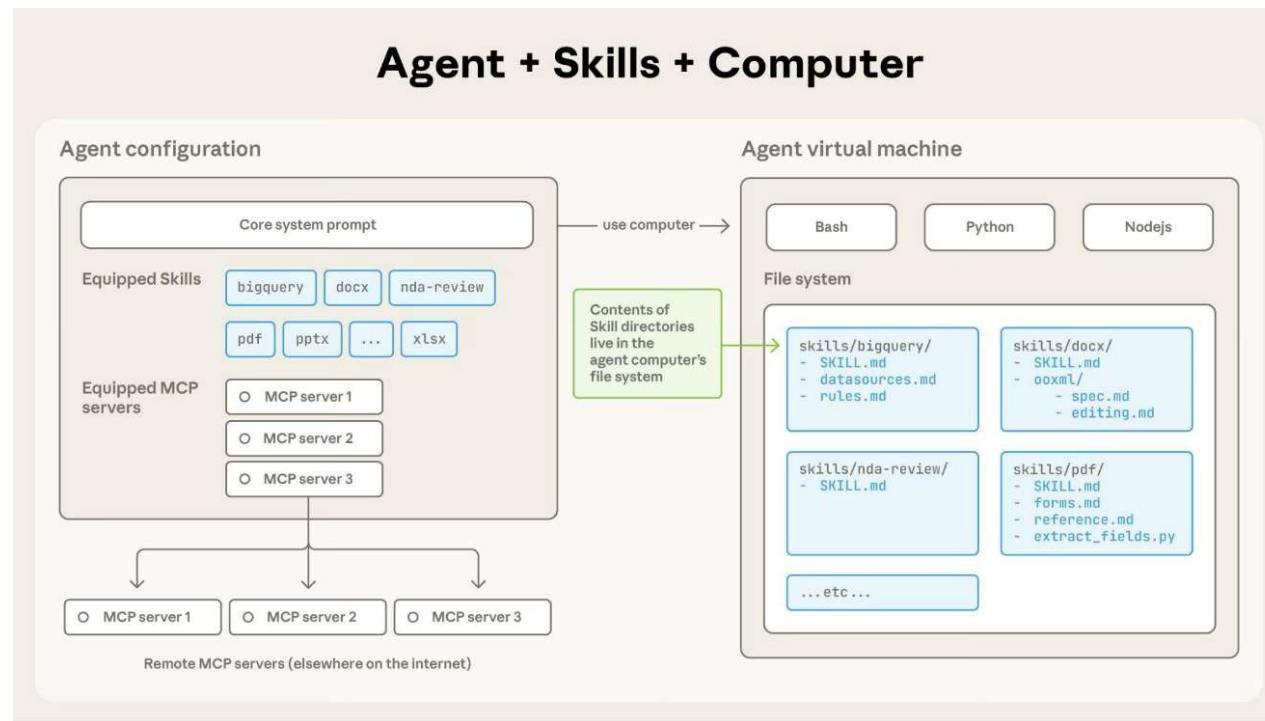
Tool Call: edit_file(...) -> remembers that the user's google account email is john@example.com

3.2 Select Context

- Pull information into context windows **as needed** to support agent performance
 - allows fine-grained control over what the LLM sees at each step
 - can improve focus, tool use, and task performance
 - requires careful filtering to avoid irrelevant content or tools
- Common mechanisms:
 1. **Scratchpad retrieval**
 2. **Memory retrieval**
 3. **Tool Search via Skills/RAG**
 4. **Knowledge Retrieval via Skills/RAG**
 5. **Demonstration Selection for In-context Learning**

Agent Skills

- Agent Skills are **folders of instructions, scripts, and resources** that agents can discover and use to do things more accurately and efficiently.
- Skill provide procedural knowledge and company-, team-, and user-specific context. Think of them as "how-to guides".

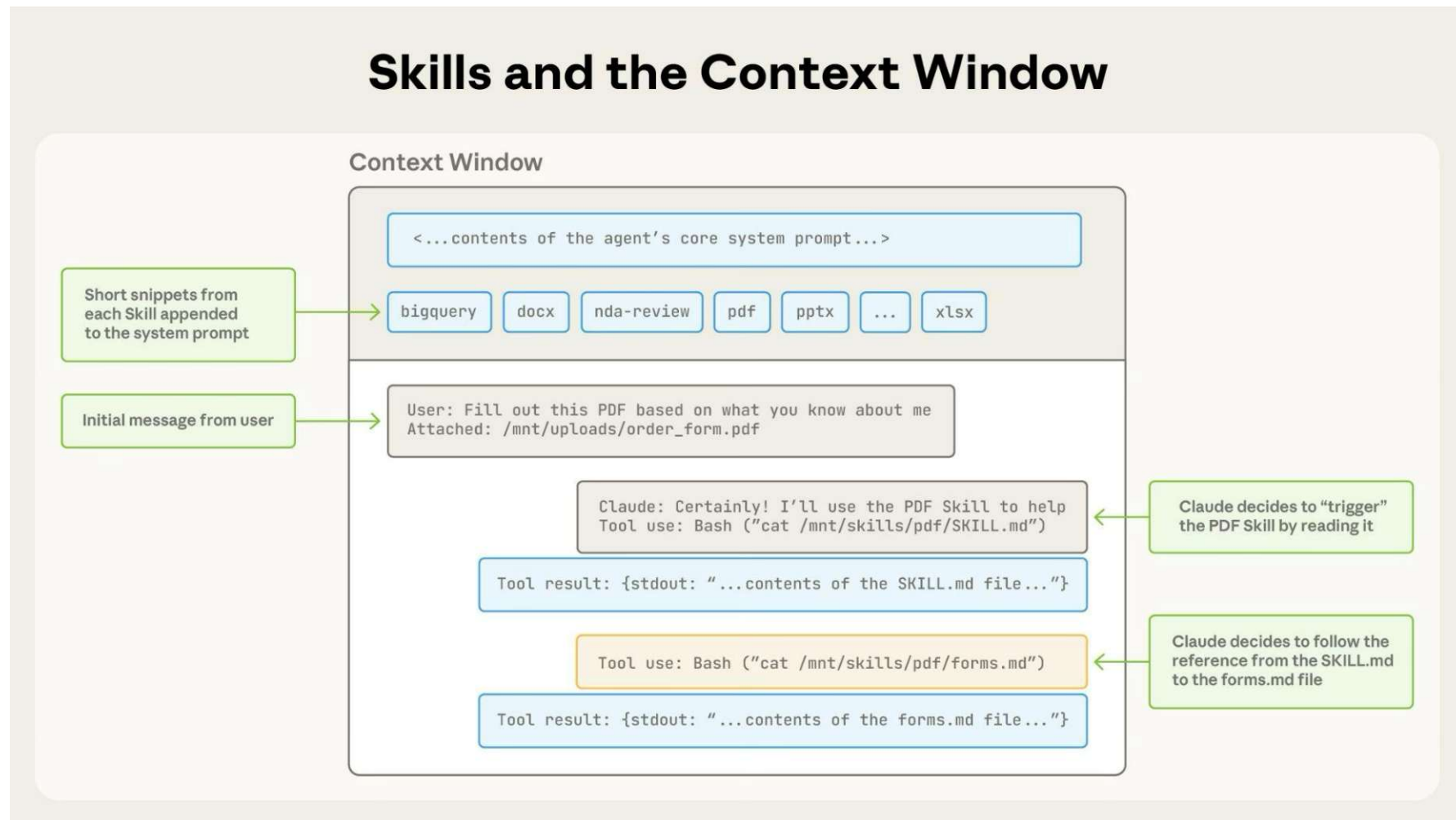


<https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview>

<https://agentskills.io/specification>

Skills are Loaded On Demand

- Every skill defines a trigger determining when it is loaded



Example: LangGraph Q&A Skill

- SKILL.md file

```
skills/  
├─ langgraph-docs  
│   └─ SKILL.md  
└─ arxiv_search  
    ├── SKILL.md  
    └─ arxiv_search.py # code for searching arXiv
```

```
---  
name: langgraph-docs  
description: Use this skill for requests related to LangGraph in order to fetch relevant  
documentation to provide accurate, up-to-date guidance.  
---  
## Overview  
This skill explains how to access LangGraph Python documentation to help answer  
questions and guide implementation.  
## Instructions  
### 1. Fetch the Documentation Index  
Use the fetch_url tool to read the following URL: https://docs.langchain.com/llms.txt  
This provides a structured list of all available documentation with descriptions.  
### 2. Select Relevant Documentation  
Based on the question, identify 2-4 most relevant documentation URLs from the index.  
Prioritize:  
- Specific how-to guides for implementation questions  
...
```

Further Skills and Their Description

Name	Description
OpenAI slides	Create and edit presentation slide decks (`.pptx`) with PptxGenJS, bundled layout helpers, and render/validation utilities. Use when tasks involve building a new PowerPoint deck, recreating slides from screenshots/PDFs/reference decks, modifying slide content.
hugging-face-model-trainer	This skill should be used when users want to train or fine-tune language models using TRL (Transformer Reinforcement Learning) on Hugging Face Jobs infrastructure. Covers SFT, DPO, GRPO and reward modeling training methods, plus GGUF conversion for local deployment.
Claude internal-comms	A set of resources to help me write all kinds of internal communications, using the formats that my company likes to use. Claude should use this skill whenever asked to write some sort of internal communications (status reports, leadership updates, 3P updates, company newsletters, FAQs, incident reports, project updates, etc.).

<https://github.com/anthropics/skills/tree/main/skills>

<https://github.com/openai/skills/tree/main/skills>

<https://clawhub.ai/>

Agent Skills

- Agent skill design guidelines
 1. Keep core skill concise and move details into referenced files
 2. Write descriptions with key terms and clear triggers for when the skill should be used
 3. Progressive disclosure:
 - Metadata always available
 - Load instructions when triggered
 - Load resources/code only when needed
 4. Structure complex tasks as explicit workflows or checklists
 5. Add feedback loops and validation steps
- List of skills for reuse
 - <https://github.com/heilcheng/awesome-agent-skills>

<https://platform.claude.com/docs/en/agents-and-tools/agent-skills/best-practices>

Activate Agent Skills in LangChain

- Enable skills in agent creation:
- Addition to system prompt:

```
agent = create_deep_agent(  
    backend=FilesystemBackend(root_dir="/Users/user/{project}"),  
    skills=["/Users/user/{project}/skills/"],  
    interrupt_on={  
        "write_file": True, # Default: approve, edit, reject  
        "read_file": False, # No interrupts needed  
        "edit_file": True # Default: approve, edit, reject  
    },  
    checkpointer=checkpointer, # Required!  
)
```

Skills System

You have access to a skills library that provides specialized capabilities and domain knowledge.

****Skills Skills****: ``/skills/`` (higher priority)

****Available Skills****

- ****web-research****: Structured approach to conducting thorough web research on any topic
-> Read ``/skills/web-research/SKILL.md`` for full instructions

****How to Use Skills (Progressive Disclosure)****

Skills follow a ****progressive disclosure**** pattern - you see their name and description above, but only read full instructions when needed:

1. ****Recognize when a skill applies****: Check if the user's task matches a skill's description
2. ****Read the skill's full instructions****: Use the path shown in the skill list above
3. ****Follow the skill's instructions****: SKILL.md contains step-by-step workflows, best practices, and examples
4. ****Access supporting files****: Skills may include helper scripts, configs, or reference docs - use absolute paths

****When to Use Skills****

- User's request matches a skill's domain (e.g., "research X" -> web-research skill)
- You need specialized knowledge or structured workflows
- A skill provides proven patterns for complex tasks

****Executing Skill Scripts****

Skills may contain Python scripts or other executable files. Always use absolute paths from the skill list.

Tool Search

- Giving an agent all tool definitions upfront
 - does not scale to thousands of tools
 - can harm performance due to irrelevant context
- Common solutions are tool discovery via
 - skills
 - or a single meta tool for tool search
- Anthropic reports 90% reduced context usage while enabling scaling to thousands of tools

<https://platform.claude.com/cookbook/tool-use-tool-search-with-embeddings>

Tool Search: Patterns

- **Retrieval-based Tool Search**
 - embed tool descriptions in dense embedding space
 - use semantic search to fetch best matching tools
 - Example: <https://platform.claude.com/cookbook/tool-use-tool-search-with-embeddings>
- **Selector-based Tool Search**
 - use a secondary LLM for selecting relevant tools from library
 - available for LangChain DeepAgent
- **Goal:** reduce irrelevant tools in context and improve model focus and accuracy

Example: Tool Search

- Tool Search in DeepAgent
- Does not change the main LLMs system prompt, instead injects relevant tools into context before the prompt is passed to the main LLM

```
from langchain.agents import create_agent
from langchain.agents.middleware import LLMToolSelectorMiddleware

agent = create_agent(
    model="gpt-4.1",
    tools=[tool1, tool2, tool3, tool4, tool5, ...],
    middleware=[
        LLMToolSelectorMiddleware(
            model="gpt-4.1-mini",
            max_tools=3,
            always_include=["search"],
        ),
    ],
)
```

SYSTEM (Selector LLM):

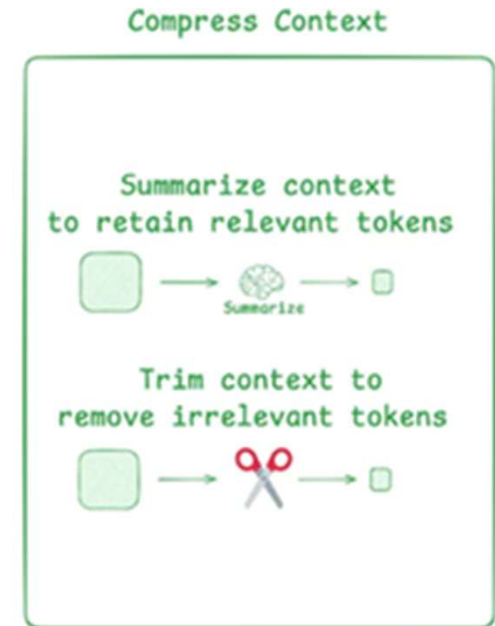
Your goal is to select the most relevant tools for answering the user's query.

IMPORTANT: List the tool names in order of relevance, with the most relevant first. If you exceed the maximum number of tools, only the first N will be used.

[HumanMessage("...user's latest message...")] (no history)

3.3 Compress Context

- Cut tokens from context to
 - retain only relevant information and avoid needle in the haystack problem
 - help manage long interaction histories and token-heavy tool calls
 - reduce context window usage, latency and cost
- Challenge: preserving important details
- Mechanisms:
 1. Context Trimming
 2. Context Summarization



Context Trimming

- **Removes** or **prunes** context instead of rewriting it
- Often based on filtering rules or heuristics, e.g.
 - Drop oldest messages
 - Drop parts of verbose tool call results
- Usually simpler and cheaper than LLM-based summarization
- **Problem 1:** may accidentally prune relevant context if rules are not designed well
- **Problem 2:** Context trimming may invalidate prompt cache
 - model call with trimmed context costs factor 10 more
 - Design decisions: To trim or not to trim? What to cache?

Example: Context Trimming

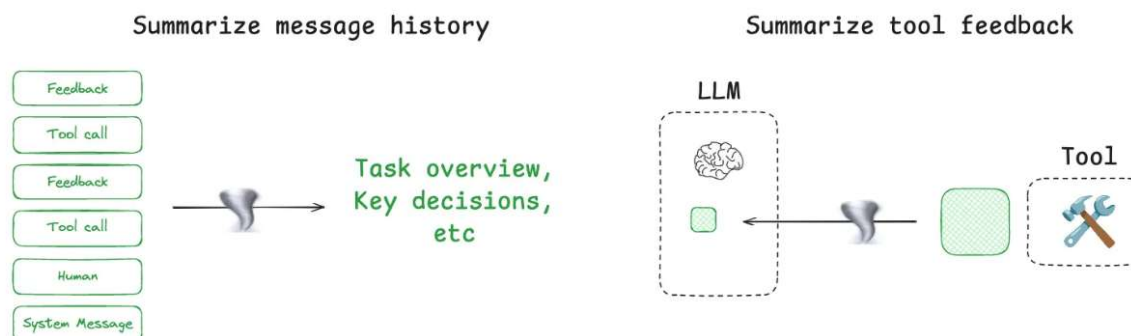
- Trimming tool calls in DeepAgent
- Set a context length threshold to trigger trimming
- Keep N most recent tool calls, throw away rest

```
from langchain.agents import create_agent
from langchain.agents.middleware import ContextEditingMiddleware, ClearToolUsesEdit

agent = create_agent(
    model="gpt-4.1",
    tools=[],
    middleware=[
        ContextEditingMiddleware(
            edits=[
                ClearToolUsesEdit(
                    trigger=100000,
                    keep=3,
                ),
            ],
        ),
    ],
)
```

Context Summarization

- Use (another) LLM to summarize long interaction histories into a shorter representation
- Common Strategies:
 - **Recursive summarization:** repeatedly summarize chunks of content, then summarize summaries to get reduced representation
 - **Hierarchical summarization:** organizes content into levels, first summarizing smaller units, then combining them into higher-level summaries that preserve structure of original information



Example: Context Summarization

- Summarization in DeepAgent
- No change to system prompt
- Prompt of summarizer LLM:

```
from langchain.agents import create_agent
from langchain.agents.middleware import SummarizationMiddleware

agent = create_agent(
    model="gpt-4.1",
    tools=[your_weather_tool, your_calculator_tool],
    middleware=[
        SummarizationMiddleware(
            model="gpt-4.1-mini",
            trigger=("tokens", 4000),
            keep=("messages", 20),
```

<primary_objective>

Your sole objective in this task is to extract the highest quality/most relevant context from the conversation history below.

</primary_objective> ...

<instructions>

The conversation history below will be replaced with the context you extract in this step.

You want to ensure that you don't repeat any actions you've already completed, so the context you extract from the conversation history should be focused on the most important information to your overall goal.

You should structure your summary using the following sections. Each section acts as a checklist - you must populate it with relevant information or explicitly state "None" if there is nothing to report for that section:

SESSION INTENT

What is the user's primary goal or request? ...

SUMMARY

Extract and record all of the most important context ...

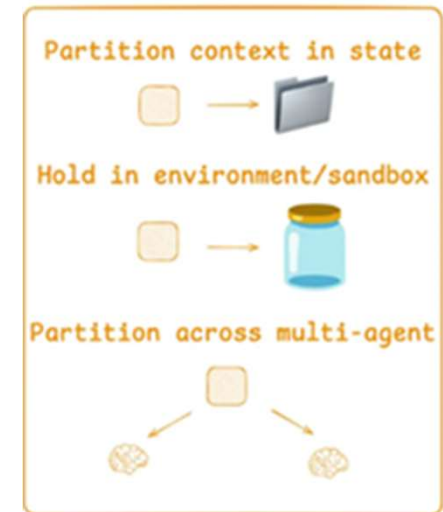
</instructions>

{...formatted conversation history...}

3.4 Isolate Context

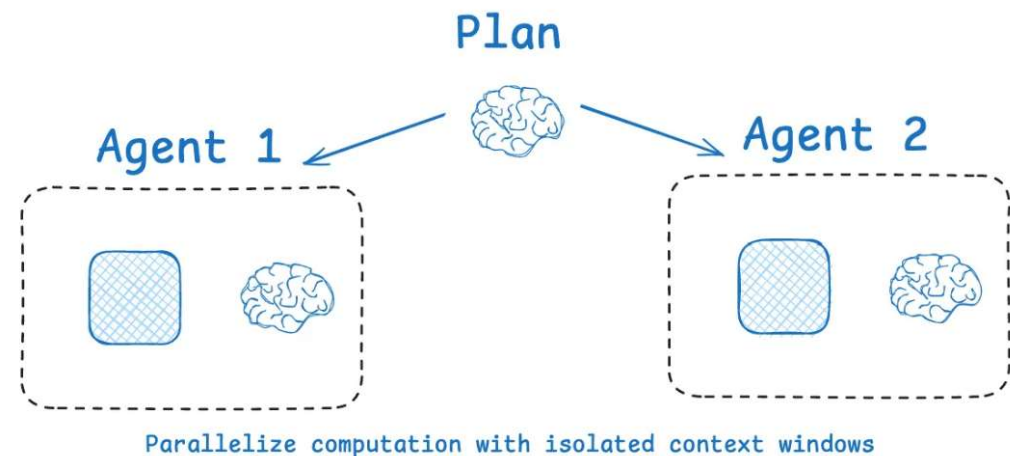
Isolate Context

- Split context into separate parts to
 - reduce overload in a single context window
 - allow different subtasks to use different instructions and tools
- Can improve model focus by keeping context narrow and relevant for current task
- Requires coordination to keep isolated contexts aligned
- Common mechanisms:
 1. **Multi-/Sub-agent systems**
 2. **Execution Environments**
 3. **Structured runtime state**



Multi-/Sub-Agents

- Split work across multiple sub-agents each with a **separate context window**
- Each agent has access to own task, tools and instructions
- Allows agents to explore different aspects of a problem
- Can outperform single agent on complex tasks
- Challenges:
 - high token usage
 - coordination overhead
 - requires careful planning



Example: Multi-/Sub-Agents

`task` (subagent spawner)

You have access to a `task` tool to launch short-lived subagents that handle isolated tasks. These agents are ephemeral — they live only for the duration of the task and return a single result.

When to use the task tool:

- When a task is complex and multi-step, and can be fully delegated in isolation
- When a task is independent of other tasks and can run in parallel
- When a task requires focused reasoning or heavy token/context usage that would bloat the orchestration
- When sandboxing improves reliability (e.g. code execution, structured searches, data formatting)
- When you only care about the output of the subagent, and not the intermediate steps (...)

Subagent lifecycle:

1. ****Spawn**** → Provide clear role, instructions, and expected output
2. ****Run**** → The subagent completes the task autonomously
3. ****Return**** → The subagent provides a single structured result
4. ****Reconcile**** → Incorporate or synthesize the result into the main thread

When NOT to use the task tool:

- If you need to see the intermediate reasoning or steps after the subagent has completed (the task is not isolated)
- If the task is trivial (a few tool calls or simple lookup)
- If delegating does not reduce token usage, complexity, or context switching
- If splitting would add latency without benefit

Important Task Tool Usage Notes to Remember

- Whenever possible, parallelize the work that you do. ...
- Remember to use the `task` tool to silo independent tasks within a multi-part objective.
- You should use the `task` tool whenever you have a complex task that will take multiple steps, ...

Available subagent types:

- weather: agent for researching weather conditions, ...
- my-custom-agent: <description> ← one line per subagent you registered

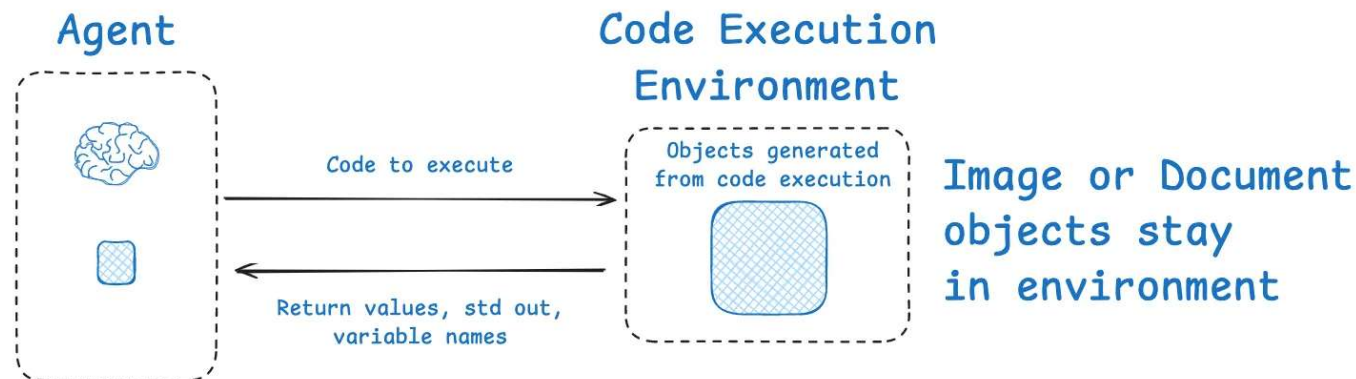
```
from langchain.tools import tool
from langchain.agents import create_agent
from deepagents.middleware.subagents import SubAgentMiddleware

@tool
def get_weather(city: str) -> str:
    """Get the weather in a city."""
    return f"The weather in {city} is sunny."

agent = create_agent(
    model="claude-sonnet-4-6",
    middleware=[
        SubAgentMiddleware(
            default_model="claude-sonnet-4-6",
            default_tools=[],
            subagents=[
                {
                    "name": "weather",
                    "description": "This subagent can get weather in c",
                    "system_prompt": "Use the get_weather tool to get",
                    "tools": [get_weather],
                    "model": "gpt-4.1",
                    "middleware": [],
                }
            ],
        )
    ],
)
```

Execution Environments

- Move part of tool calls outside the LLMs context window into an execution environment (e.g. python interpreter for coding agents)
- Tool calls run in sandbox and only selected results are returned to the model
- Useful for isolating token-heavy objects like files, images or intermediate outputs



Structured Runtime State

- Stores information in a runtime state object instead of directly in context window
- State can be organized with **schema** such as typed field for
 - Messages
 - Plans
 - Results
- Only selected fields need to be exposed at each step
- Supports fine-grained control over what model can access

Summary: Context Engineering

- Long agent trajectories with tool definitions/calls and corresponding results fill up the context window quickly
 - loss of accuracy due to **context rot**
 - **high cost and latency**
- Context Engineering is the craft of giving the LLM **the right information at the right time** (in the right order)
 - effective agentic systems combine system prompts and agent harness mechanisms to provide the right context for the next call via Write/Select/Compress/Isolate context techniques
 - prompt caching helps increase efficiency by exploiting order and reusing stored calculations
- Well-executed context engineering improves quality, efficiency, and scalability of LLM agents

See you next week!

- Next time:
 - Team formation and project kickoff
 - Teams of 5 students work on the application of LLM agents to a real use-case
 - Please use [this google sheet](#) for finding team mates and entering your final (sub-)teams.
 - We will merge teams with less than 5 students into full teams and let you know the final teams next week!

