Context Engineering - Short-Term Memory Management with Sessions from OpenAI Agents SDK

 Emre Okcular

AI agents often operate in **long-running, multi-turn interactions**, where keeping the right balance of **context** is critical. If too much is carried forward, the model risks distraction, inefficiency, or outright failure. If too little is preserved, the agent loses coherence.

Here, context refers to the total window of tokens (input + output) that the model can attend to at once. For GPT-5, this capacity is up to 272k input tokens and 128k output tokens but even such a large window can be overwhelmed by uncurated histories, redundant tool results, or noisy retrievals. This makes context management not just an optimization, but a necessity.

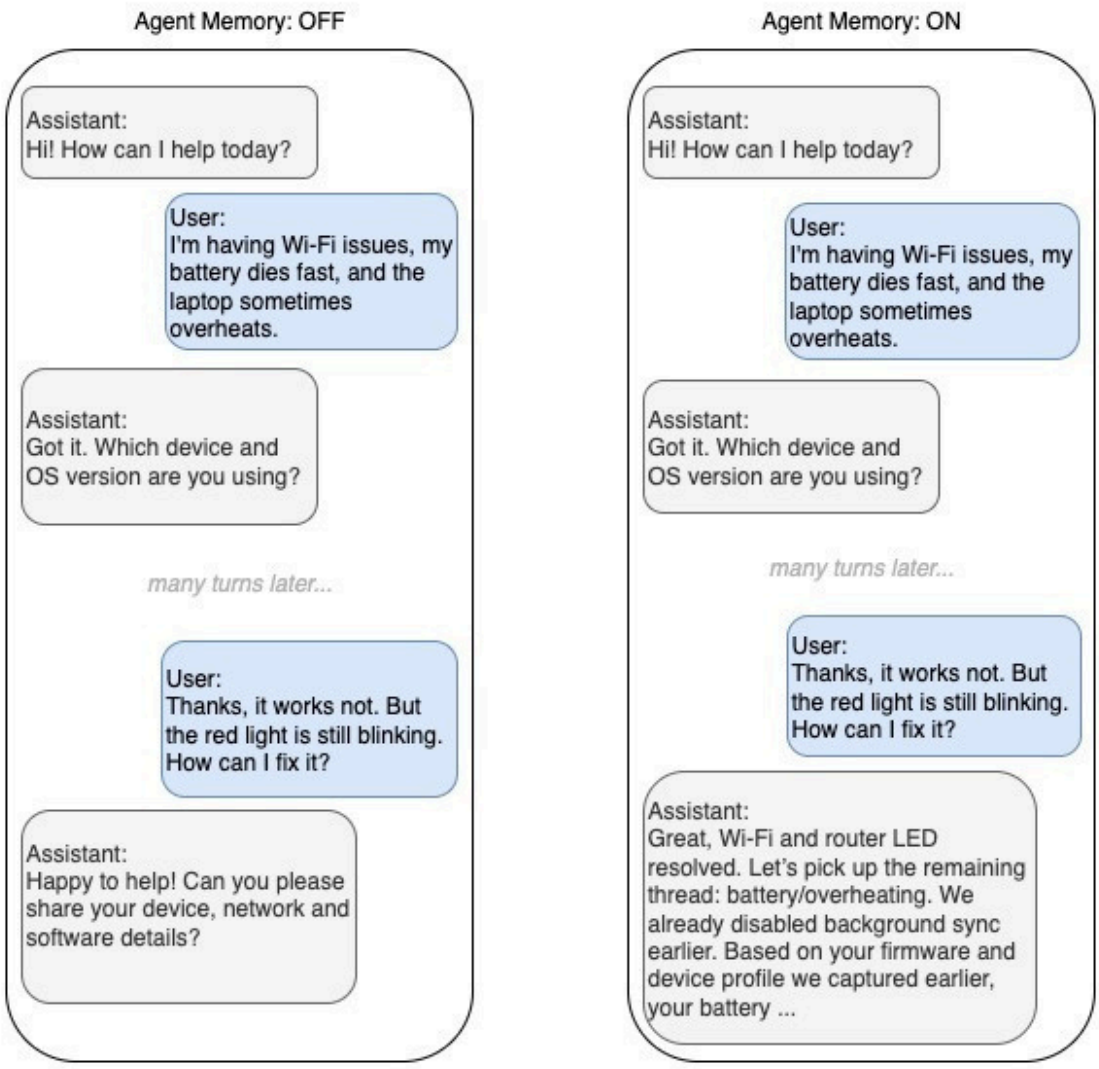
In this cookbook, we'll explore how to **manage context effectively using the**

Session **object from the** OpenAI Agents SDK, focusing on two proven context management techniques—**trimming** and **compression**—to keep agents fast, reliable, and cost-efficient.

Why Context Management Matters

Sustained coherence across long threads – Keep the agent anchored to the latest user goal without dragging along stale details. Session-level trimming and summaries prevent "yesterday's plan" from overriding today's ask.

- **Higher tool-call accuracy** – Focused context improves function selection and argument filling, reducing retries, timeouts, and cascading failures during multi-tool runs.
- **Lower latency & cost** – Smaller, sharper prompts cut tokens per turn and attention load.
- **Error & hallucination containment** – Summaries act as “clean rooms” that correct or omit prior mistakes; trimming avoids amplifying bad facts (“context poisoning”) turn after turn.
- **Easier debugging & observability** – Stable summaries and bounded histories make logs comparable: you can diff summaries, attribute regressions, and reproduce failures reliably.
- **Multi-issue and handoff resilience** – In multi-problem chats, per-issue mini-summaries let the agent pause/resume, escalate to humans, or hand off to another agent while staying consistent.



The OpenAI Responses API includes **basic memory support** through built-in state and message chaining with `previous_response_id`.

You can continue a conversation by passing the prior response's `id` as `previous_response_id`, or you can manage context manually by collecting outputs into a list and resubmitting them as the `input` for the next response.

What you don't get is **automatic memory management**. That's where the **Agents SDK** comes in. It provides session memory on top of Responses, so you no longer need to manually append `response.output` or track IDs yourself. The session becomes the **memory object**: you simply call `session.run("...")` repeatedly, and the SDK handles context length, history, and continuity—making it far easier to build coherent, multi-turn agents.

Real-World Scenario

We'll ground the techniques in a practical example for one of the common long-running tasks, such as:

- **Multi-turn Customer Service Conversations** In extended conversations about tech products—spanning both hardware and software—customers often surface multiple issues over time. The agent must stay consistent and goal-focused while retaining only the essentials rather than hauling along every past detail.

Techniques Covered

To address these challenges, we introduce two separate concrete approaches using OpenAI Agents SDK:

- **Context Trimming** – dropping older turns while keeping the last N turns.
    - **Pros**
        - **Deterministic & simple:** No summarizer variability; easy to reason about state and to reproduce runs.
        - **Zero added latency:** No extra model calls to compress history.
        - **Fidelity for recent work:** Latest tool results, parameters, and edge cases stay verbatim—great for debugging.

- **Lower risk of "summary drift":** You never reinterpret or compress facts.

Cons

- **Forgets long-range context abruptly:** Important earlier constraints, IDs, or decisions can vanish once they scroll past N.
- **User experience "amnesia":** Agent can appear to "forget" promises or prior preferences midway through long sessions.
- **Wasted signal:** Older turns may contain reusable knowledge (requirements, constraints) that gets dropped.
- **Token spikes still possible:** If a recent turn includes huge tool payloads, your last-N can still blow up the context.

- **Best when**
  - Your tasks in the conversation is indepentent from each other with non-overlapping context that does not reuqire carrying previous details further.
  - You need predictability, easy evals, and low latency (ops automations, CRM/API actions).
  - The conversation's useful context is local (recent steps matter far more than distant history).

- **Context Summarization** – compressing prior messages(assistant, user, tools, etc.) into structured, shorter summaries injected into the conversation history.
  - **Pros**
    - **Retains long-range memory compactly:** Past requirements, decisions, and rationales persist beyond N.
    - **Smoother UX:** Agent "remembers" commitments and constraints across long sessions.
    - **Cost-controlled scale:** One concise summary can replace hundreds of turns.
    - **Searchable anchor:** A single synthetic assistant message becomes a stable "state of the world so far."

Cons

- **Summarization loss & bias:** Details can be dropped or misweighted; subtle constraints may vanish.
- **Latency & cost spikes:** Each refresh adds model work (and potentially tool-trim logic).
- **Compounding errors:** If a bad fact enters the summary, it can **poison** future behavior ("context poisoning").
- **Observability complexity:** You must log summary prompts/outputs for auditability and evals.

- **Best when**
  - You have use cases where your tasks needs context collected accross the flow such as planning/coaching, RAG-heavy analysis, policy Q&A.
  - You need continuity over long horizons and carry the important details further to solve related tasks.
  - Sessions exceed N turns but must preserve decisions, IDs, and constraints reliably.

Quick comparison

Dimension	Trimming (last-N turns)	Summarizing (older → generated summary)
Latency / Cost	Lowest (no extra calls)	Higher at summary refresh points
Long-range recall	Weak (hard cut-off)	Strong (compact carry-forward)
Risk type	Context loss	Context distortion/poisoning
Observability	Simple logs	Must log summary prompts/outputs
Eval stability	High	Needs robust summary evals
Best for	Tool-heavy ops, short workflows	Analyst/concierge, long threads

Prerequisites

Before running this cookbook, you must set up the following accounts and complete a few setup actions. These prerequisites are essential to interact with the APIs used in this project.

Step0: OpenAI Account and `OPENAI_API_KEY`

- **Purpose:**
  You need an OpenAI account to access language models and use the Agents SDK featured in this cookbook.

- **Action:**
  ~~Sign up for an OpenAI account~~ if you don't already have one. Once you have an account, create an API key by visiting the ~~OpenAI API Keys page~~.

Before running the workflow, set your environment variables:

```
# Your openai key
os.environ["OPENAI_API_KEY"] = "sk-proj-..."
```

Alternatively, you can set your OpenAI API key for use by the agents via the `set_default_openai_key` function by importing agents library .

```
from agents import set_default_openai_key
set_default_openai_key("YOUR_API_KEY")
```

Step1: Install the Required Libraries

Below we install the `openai-agents` library (OpenAI Agents SDK)

```
%pip install openai-agents nest_asyncio
```

```
from openai import OpenAI

client = OpenAI()
```

```
from agents import set_tracing_disabled
set_tracing_disabled(True)
```

Let's test the installed libraries by defining and running an agent.

```
import asyncio
from agents import Agent, Runner


agent = Agent(
    name="Assistant",
    instructions="Reply very concisely.",
)

result = await Runner.run(agent, "Tell me why it is important to evaluat
print(result.final_output)
```

```
Evaluating AI agents ensures reliability, safety, ethical alignment
```

Define Agents

We can start by defining the necessary components from Agents SDK Library. Instructions added based on the use case during agent creation.

Customer Service Agent

```
support_agent = Agent(
    name="Customer Support
    Assistant", model="gpt-5",
    instructions(
   "You are a patient, step-by-step IT support assistant. " "Your
   role is to help customers troubleshoot and resolve issues
   "Guidelines:\n" "- Be concise and use numbered steps where
   possible.\n" "- Ask only one focused, clarifying question at a
   time before su "- Track and remember multiple issues across
   the conversation; u "- When a problem is resolved, briefly
   confirm closure before mo
    )
)
```

Context Trimming

Implement Custom Session Object

We are using Session object from OpenAI Agents Python SDK. Here's a `TrimmingSession` implementation that **keeps only the last N turns** (a "turn" = one user message and everything until the next user message—including the assistant reply and any tool calls/results). It's in-memory and trims automatically on every write and read.

```
from __future__ import annotations

import asyncio
from collections import deque
from typing import Any, Deque, Dict, List, cast

from agents.memory.session import SessionABC from agents.items import# dict-like item
TResponseInputItem
ROLE_USER = "user"


def _is_user_msg(item: TResponseInputItem) -> bool:
    """Return True if the item represents a user message."""
    # Common dict-shaped messages
    if isinstance(item, dict):
        role = item.get("role")
        if role is not None:
            return role == ROLE_USER
        # Some SDKs: {"type": "message", "role": "..."}
        if item.get("type") == "message":
            return item.get("role") == ROLE_USER
    # Fallback: objects with a .role attr
    return getattr(item, "role", None) == ROLE_USER


class TrimmingSession(SessionABC):
    """
    Keep only the last N *user turns* in memory.

  A turn = a user message and all subsequent items (assistant/tool ca
  up to (but not including) the next user message.
  """

    def __init__(self, session_id: str, max_turns: int = 8):
        self.session_id = session_id
        self.max_turns = max(1, int(max_turns))
```

```
        self._items: Deque[TResponseInputItem] = deque() # chronologica
        self._lock = asyncio.Lock()

    # ---- SessionABC API ----

    async def get_items(self, limit: int | None = None) ->List[TRespon
        """Return history trimmed to the last N user turns (optionally
        async with self._lock:
            trimmed = self._trim_to_last_turns(list(self._items))
            return trimmed[-limit:] if (limit is not None and limit >= 0

    async def add_items(self, items: List[TResponseInputItem]) -> None:
        """Append new items, then trim to last N user turns."""
        if not items:
            return
        async with self._lock:
            self._items.extend(items)
            trimmed = self._trim_to_last_turns(list(self._items))
            self._items.clear()
            self._items.extend(trimmed)

    async def pop_item(self) -> TResponseInputItem | None:
        """Remove and return the most recent item (post-trim)."""
        async with self._lock:
            return self._items.pop() if self._items else None

    async def clear_session(self) -> None:
        """Remove all items for this session."""
        async with self._lock:
            self._items.clear()

    # ---- Helpers ----

    def _trim_to_last_turns(self, items: List[TResponseInputItem]) -> L
        """
        Keep only the suffix containing the last `max_turns` user messag
        the earliest of those user messages.

        If there are fewer than `max_turns` user messages (or none), kee
        """
        if not items:
            return items

        count = 0
        start_idx = 0   # default: keep all if we never reach max_turns

        # Walk backward; when we hit the Nth user message, mark its inde
        for i in range(len(items) - 1, -1, -1):
            if _is_user_msg(items[i]):
                count += 1
                if count == self.max_turns:
                    start_idx = i
```

```
                    break

        return items[start_idx:]

    #----Optionalconvenience API ----

    async def set_max_turns(self, max_turns: int) -> None:
        async with self._lock:
            self.max_turns = max(1, int(max_turns))
            trimmed = self._trim_to_last_turns(list(self._items))
            self._items.clear()
            self._items.extend(trimmed)

    async def raw_items(self) -> List[TResponseInputItem]:
        """Returntheuntrimmed in-memory log (for debugging)."""
        async with self._lock:
            return list(self._items)
```

Let's define the custom session object we implemented with max_turns=3.

```
# Keep only the last 8 turns (user + assistant/tool interactions)
session = TrimmingSession("my_session", max_turns=3)
```

How to choose the right `max_turns`?

Determining this parameter usually requires experimentation with your conversation history. One approach is to extract the total number of turns across conversations and analyze their distribution. Another option is to use an LLM to evaluate conversations—identifying how many tasks or issues each one contains and calculating the average number of turns needed per issue.

```
message = "There is a red light blinking on my laptop."
```

```
result = await Runner.run(
    support_agent,
    message,
    session=session
)
```

```
history = await session.get_items()
```

```
history
```

```
[{'content': 'There is a red light blinking on my laptop.', 'role':
  {'id': 'rs_68be66229c008190aa4b3c5501f397080fdfa41323fb39cb',
 'summary': [],
'type': 'reasoning',
'content': []},
{'id': 'msg_68be662f704c8190969bdf539701a3e90fdfa41323fb39cb',
'content': [{'annotations': [],
      'text': 'A blinking red light usually indicates a power/battery
      'type': 'output_text',
      'logprobs': []}],
 'role': 'assistant',
'status': 'completed',
'type': 'message'}]
```

```
# Example flow await session.add_items([{"role": "user", "content": "I am
using a macbo await session.add_items([{"role": "assistant", "content": "I
see. Let's await session.add_items([{"role": "user", "content": "Firmware
v1.0.3; await session.add_items([{"role": "assistant", "content": "Could
you ple await session.add_items([{"role": "user", "content": "Reset done;
error await session.add_items([{"role": "assistant", "content": "Leave it
on await session.add_items([{"role": "user", "content": "Yes, I see error 4
await session.add_items([{"role": "assistant", "content": "Do you see it #
At this point, with max_turns=3, everything *before* the earliest of t #
messages is summarized into a synthetic pair, and the last 3 turns rem



history = await session.get_items()
# Pass `history` into your agent runner / responses call as the conversa
```

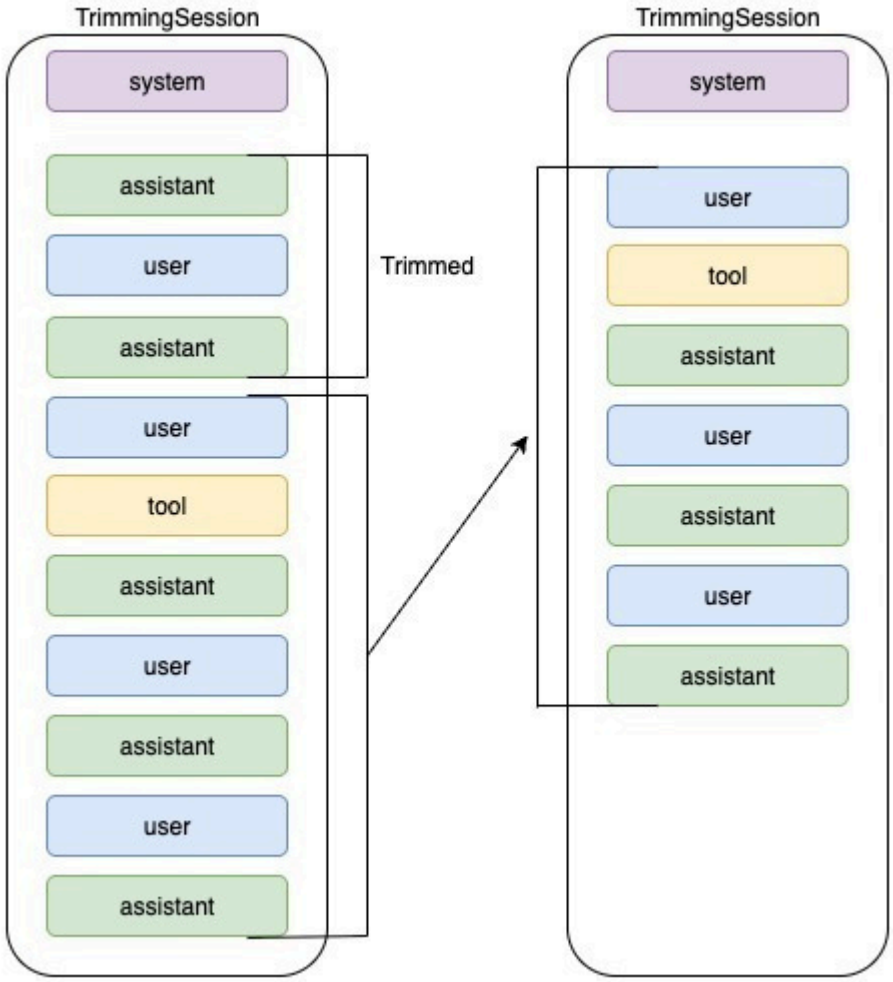
```
len(history)
```

```
6
```

```
history
```

```
[{'role': 'user', 'content': 'Firmware v1.0.3; still failing.'},
{'role': 'assistant', 'content': 'Could you please try a factory r
{'role': 'user', 'content': 'Reset done; error 42 now.'},
{'role': 'assistant',
'content': 'Leave it on charge for 30 minutes in case the battery
 {'role': 'user', 'content': 'Yes, I see error 404 now.'},
 {'role': 'assistant',
'content': 'Do you see it on the browser while accessing a websit
```

Below, you can see how the trimming session works for max_turns=3.



What counts as a "turn"

- A **turn** = one **user** message **plus everything that follows it** (assistant replies, reasoning, tool calls, tool results) **until the next user message**.

When trimming happens

- On **write**: `add_items(...)` appends the new items, then immediately trims the stored history.
- On **read**: `get_items(...)` returns a **trimmed** view (so even if you bypassed a write, reads won't leak old turns).

How it decides what to keep

1. Treat any item with `role == "user"` as a **user message** (via `_is_user_msg`).
2. Scan the history **backwards** and collect the indices of the last **N** user messages (`max_turns`).
3. Find the **earliest** index among those N user messages.
4. **Keep everything from that index to the end**; drop everything before it.

That preserves each complete turn boundary: if the earliest kept user message is at index `k`, you also keep all assistant/tool items that came after `k`.

Tiny example

History (old → new):

```
0: user("Hi") 1: assistant("Hello!") 2: tool_call("lookup") 3: tool_result("…") 4: user("It didn't work") 5: assistant("Try rebooting") 6: user("Rebooted, now error 42") 7: assistant("On it")
```

With `max_turns = 2`, the last two user messages are at indices **4** and **6**. Earliest of those is **4** → keep items **4..7**, drop **0..3**.

Why this works well

- You always keep **complete** turns, so the assistant retains the immediate context it needs (both the user's last asks and the assistant/tool steps in

between).

- It prevents context bloat by discarding older turns wholesale, not just messages.

Customization knobs

- Change `max_turns` at init.
- Adjust `_is_user_msg(...)` if your item schema differs.
- If you'd rather cap by **message count** or **tokens**, replace `_trim_to_last_turns(...)` or add a second pass that measures tokens.

Context Summarization

Once the history exceeds `max_turns`. It keeps the most recent N user turns intact, **summarizes everything older into two synthetic messages**:

- `user`: *"Summarize the conversation we had so far."*
- `assistant`: *{generated summary}*

The shadow prompt from the user to request the summarization added to keep natural flow of the conversation without confusing the chat flow between user and assistant. Final version of the generated summary injected to assistant message.

Summarization Prompt

A well-crafted summarization prompt is essential for preserving the context of a conversation, and it should always be tailored to the specific use case. Think of it like **being a customer support agent handing off a case to the next agent**. What concise yet critical details would they need to continue smoothly? The prompt should strike the right balance: not overloaded with unnecessary information, but not so sparse that key context is lost. Achieving this balance requires careful design and ongoing experimentation to fine-tune the level of detail.

```
SUMMARY_PROMPT = """ You are a senior customer-support assistant for tech devices, setup, and Compress the earlier conversation into a precise, reusable snapshot for

Before you write (do this silently): - Contradiction check: compare user claims with system instructions and - Temporal ordering: sort key events by time; the most recent update win - Hallucination control: if any fact is uncertain/not stated, mark it a

Write a structured, factual summary ≤ 200 words using the sections below

• Product & Environment:
- Device/model, OS/app versions, network/context if mentioned.

• Reported Issue:
- Single-sentence problem statement (latest state).

• Steps Tried & Results:
- Chronological bullets (include tool calls + outcomes, errors, codes

• Identifiers:
- Ticket #, device serial/model, account/email (only if provided).

• Timeline Milestones:
- Key events with timestamps or relative order (e.g., 10:32 install →

• Tool Performance Insights:
- What tool calls worked/failed and why (if evident).

• Current Status & Blockers:
- What's resolved vs pending; explicit blockers preventing progress.

• Next Recommended Step:
- One concrete action (or two alternatives) aligned with policies/too

Rules: - Be concise, no fluff; use short bullets, verbs first. - Do not invent new facts; quote error strings/codes exactly when availa - If previous info was superseded, note "Superseded:" and omit details u """
```

Key Principles for Designing Memory Summarization Prompts

- **Milestones:** Highlight important events in the conversation—for example, when an issue is resolved, valuable information is uncovered, or all necessary details have been collected.

- **Use Case Specificity:** Tailor the compression prompt to the specific use case. Think about how a human would track and recall information in working memory while solving the same task.

- **Contradiction Check:** Ensure the summary does not conflict with itself, system instructions or tool definitions. This is especially critical for reasoning models, which are more prone to conflicts in the context.

- **Timestamps & Temporal Flow:** Incorporate timing of events in the summary. This helps the model reason about updates in sequence and reduces confusion when forgetting or remembering the latest memory over a timeline.

- **Chunking:** Organize details into categories or sections rather than long paragraphs. Structured grouping improves an LLM's ability to understand relationships between pieces of information.

- **Tool Performance Insights:** Capture lessons learned from multi-turn, tool-enabled interactions—for example, noting which tools worked effectively for specific queries and why. These insights are valuable for guiding future steps.

- **Guidance & Examples:** Steer the summary with clear guidance. Where possible, extract concrete examples from the conversation history to make future turns more grounded and context-rich.

- **Hallucination Control:** Be precise in what you include. Even minor hallucinations in a summary can propagate forward, contaminating future context with inaccuracies.

- **Model Choice:** Select a summarizer model based on use case requirements, summary length, and tradeoffs between latency and cost. In some cases, using the same model as the AI agent itself can be advantageous.

```
class LLMSummarizer:
    def __init__(self, client, model="gpt-4o", max_tokens=400, tool_trim
        self.client = client
        self.model = model
        self.max_tokens = max_tokens
        self.tool_trim_limit = tool_trim_limit

    async def summarize(self, messages: List[Item]) -> Tuple[str, str]:
```

```
        """
        Create a compact summary from `messages`.

        Returns:
            Tuple[str,str]:Theshadowuserline to keep dialog natura
            and the model-generated summary text.
        """
        user_shadow = "Summarizethe conversation we had so far."
        TOOL_ROLES = {"tool", "tool_result"}

        def to_snippet(m: Item) -> str | None:
            role = (m.get("role") or "assistant").lower()
            content = (m.get("content") or "").strip()
            if not content:
            return None
            #Trimverbosetooloutputsto    keep prompt compact
            if  role  in  TOOL_ROLESand len(content) > self.tool_trim_lim
                content= content[: self.tool_trim_limit] + " …"
            return f"{role.upper()}: {content}"

        # Build compact, trimmed history
        history_snippets = [s  for  min  messages if (s := to_snippet(m))

        prompt_messages = [
            {"role": "system", "content": SUMMARY_PROMPT},
            {"role": "user", "content": "\n".join(history_snippets)},
        ]

        resp = await asyncio.to_thread(
            self.client.responses.create,
            model=self.model,
            input=prompt_messages,
            max_output_tokens=self.max_tokens,
        )

        summary = resp.output_text
        await asyncio.sleep(0)  # yield control
        return user_shadow, summary
```

```
import asyncio
from collections import deque
from typing import Optional, List, Tuple, Dict, Any

Record = Dict[str, Dict[str, Any]]     # {"msg": {...}, "meta": {...}}

class SummarizingSession:
    """
    Session that keeps only the last N *user turns* verbatim and summar
```

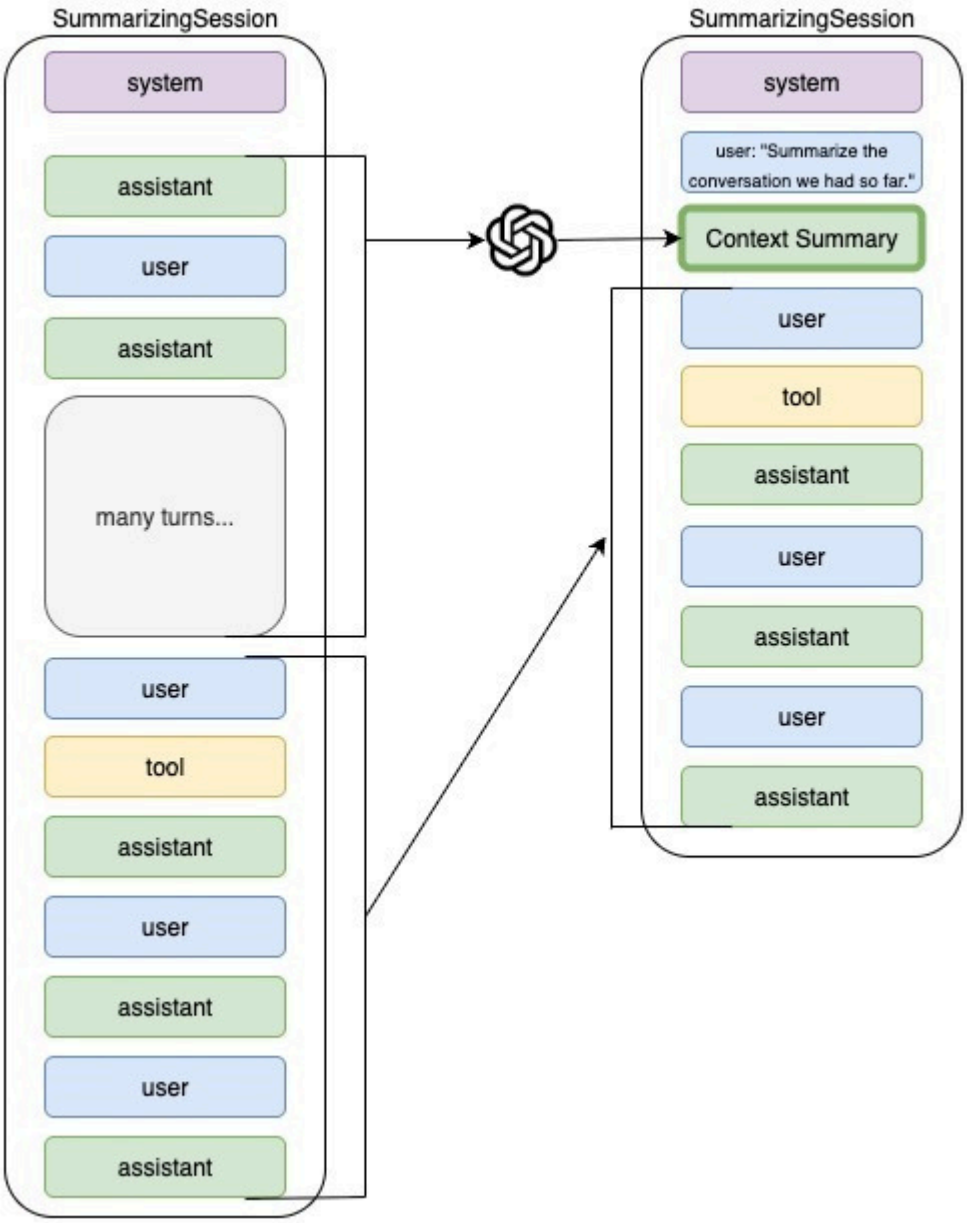
```
 - A *turn* starts at a real user message and includes everything unt
- When the number of real user turns exceeds `context_limit`, everyt
  of the last `keep_last_n_turns` user-turn starts is summarized int
- Stores full records (message + metadata). Exposes:
    ·get_items():              model-
    ·get_full_history():       safemessagesonly(nometadata)
"""                            [{"message":msg, "metadata": meta},.

# Only these keys are ever sent to the model; the rest live in metad
_ALLOWED_MSG_KEYS = {"role", "content", "name"}

def __init__(
self,
keep_last_n_turns: int = 3,
context_limit: int = 3,
summarizer:Optional["Summarizer"] = None,
session_id:Optional[str] = None,
):
    assert context_limit >= 1
    assert keep_last_n_turns >= 0
    assert keep_last_n_turns <= context_limit, "keep_last_n_turns sh

    self.keep_last_n_turns = keep_last_n_turns
    self.context_limit = context_limit
    self.summarizer = summarizer
    self.session_id = session_id or "default"

    self._records:deque[Record] = deque()
    self._lock = asyncio.Lock()

# --------- public API used by your runner ---------
async def get_items(self, limit:Optional[int] = None)->List[Dict
    """Return model-safe messages only (no metadata)."""
    async with self._lock:
        data = list(self._records)
    msgs = [self._sanitize_for_model(rec["msg"]) for rec in data]
    return msgs[-limit:] if limit else msgs

async def add_items(self, items:List[Dict[str,Any]])-> None:
    """Append new items and, if needed, summarize older turns."""
    # 1) Ingest items
    async with self._lock:
        for it in items:
            msg,meta = self._split_msg_and_meta(it)
            self._records.append({"msg":msg, "meta":meta})

        need_summary,boundary = self._summarize_decision_locked()

    # 2) No summarization needed → just normalize flags and exit
    if not need_summary:
        async with self._lock:
            self._normalize_synthetic_flags_locked()
```

```
            return

        #3)Preparesummaryprefix(model-safecopy)outside the lock
        async with self._lock:
            snapshot = list(self._records)
            prefix_msgs = [r["msg"] for r in snapshot[:boundary]]

        user_shadow,assistant_summary = await self._summarize(prefix_m

        # 4) Re-check and apply summary atomically
        async with self._lock:
            still_need,new_boundary = self._summarize_decision_locke
            if not still_need:
                self._normalize_synthetic_flags_locked()
                return

            snapshot = list(self._records)
            suffix  = snapshot[new_boundary:]#keep-last-N turns live h

            # Replace with: synthetic pair + suffix
            self._records.clear()
            self._records.extend([
                {
                    "msg":{"role": "user", "content":user_shadow},
                    "meta":{
                        "synthetic": True,
                        "kind": "history_summary_prompt",
                        "summary_for_turns":f"<  all  before idx {new_bou
                    },
                },
                {
                    "msg":{"role":   "assistant",  "content": assistant_s
                    "meta":{
                        "synthetic": True,
                        "kind": "history_summary",
                        "summary_for_turns":f"<  all  before idx {new_bou
                    },
                },
            ])
            self._records.extend(suffix)

            #Ensureallrealuser/assistantmessagesexplicitly have sy
            self._normalize_synthetic_flags_locked()

    async def pop_item(self)->Optional[Dict[str,Any]]:
        """Pop the latest message (model-safe), if any."""
        async with self._lock:
            if not self._records:
                return None
            rec = self._records.pop()
            return dict(rec["msg"])
```

```
    async def clear_session(self) -> None:
        """Remove all records."""
        async with self._lock:
            self._records.clear()

    def set_max_turns(self, n: int) -> None:
        """
        Back-compat shim for old callers: update `context_limit`
        and clamp `keep_last_n_turns` if needed.
        """
        assert n >= 1
        self.context_limit = n
        if self.keep_last_n_turns > self.context_limit:
        self.keep_last_n_turns = self.context_limit

    # Full history (debugging/analytics/observability)

    async def get_full_history(self,    limit:Optional[int] = None) -> Li
        """
        Return combined history entries in the shape:
        {"message":{role,content[,       name]},"metadata": {...}}
        ThisisNOTsenttothemodel;       for  logs/UI/debugging only.
        """
        async with self._lock:
            data = list(self._records)
        out = [{"message": dict(rec["msg"]),  "metadata": dict(rec["meta"
        return out[-limit:] if limit else out

    # Back-compat alias
    async def get_items_with_metadata(selfjimit:Optional[int] = None
        return await self.get_full_history(limit)

    # Internals

    def _split_msg_and_meta(self,it:   Dict[str,Any]) -> Tuple[Dict[str
        """
        Split input into (msg, meta):
          -msgkeepsonly_ALLOWED_MSG_KEYS;ifrole/content missing, d
          -everythingelsegoesunder   meta(including nested "metadata"
          -defaultsynthetic=Falseforrealuser/assistant unless expl
        """    msg    =    {k:    v    for    k,    v    in    it.items()    if    k    in
        self._ALLOWED_MSG_KEY not in self._ALLOWED_M
        extra = {k:v    for k,v    in it.items()   if   k
        meta = dict(extra.pop("metadata", {}))
        meta.update(extra)

        msg.setdefault("role", "user")
        msg.setdefault("content", str(it))

        role = msg.get("role")
        if  role in  ("user",   "assistant")  and"synthetic" not in meta:
            meta["synthetic"] = False
```

```
        return msg, meta

    @staticmethod
    def _sanitize_for_model(msg:Dict[str,Any])->    Dict[str, Any]:
        """Drop anything not allowed in model calls."""
        return {k: v for k, v in msg.items() if k in SummarizingSession

    @staticmethod
    def _is_real_user_turn_start(rec:Record)-> bool:
        """Trueifrecordstartsa      *real*user   turn(role=='user' and no
        return (
            rec["msg"].get("role") == "user"
            and not rec["meta"].get("synthetic", False)
        )

    def _summarize_decision_locked(self) ->Tuple[bool, int]:
        """
        Decidewhethertosummarizeandcomputetheboundary index.

        Returns:
            (need_summary, boundary_idx)

        If need_summary:
          ·boundary_idxistheearliestindex      amongthelast `keep_last
            *real* user-turn starts.
          ·Everythingbeforeboundary_idx  becomesthe summary prefix.
        """
        user_starts:List[int] = [
            i for i,rec in enumerate(self._records) if self._is_real_u
        ]
        real_turns = len(user_starts)

        # Not over the limit → nothing to do
        if real_turns <= self.context_limit:
            return False, -1

        # Keep zero turns verbatim → summarize everything
        if self.keep_last_n_turns == 0:
            return True, len(self._records)

        #Otherwise,keepthelast    Nturns;summarizeeverything before
        if len(user_starts) < self.keep_last_n_turns:
            return False,  -1   #defensive(shouldn't   happen given the ea

        boundary = user_starts[-self.keep_last_n_turns]

        #Ifthereisnothingbeforeboundary,      thereisnothing to summa
        if boundary <= 0:
            return False, -1

        return True,boundary
```

```
    def _normalize_synthetic_flags_locked(self) -> None:
        """Ensureallreal user/assistant records explicitly carry synth
        for rec in self._records:
            role = rec["msg"].get("role")
            if role in ("user", "assistant") and "synthetic" not in rec
                rec["meta"]["synthetic"] = False

    async def _summarize(self, prefix_msgs: List[Dict[str, Any]]) -> Tup
        """
        Asktheconfigured summarizer to compress the given prefix.
        Usesmodel-safemessages only. If no summarizer is configured,
        returnsagraceful fallback.
        """
        if not self.summarizer:
            return ("Summarize the conversation we had so far.", "Summa
        clean_prefix = [self._sanitize_for_model(m) for m in prefix_msg
        return await self.summarizer.summarize(clean_prefix)
```



High-level idea

- **A turn** = one **real user** message **plus everything that follows it** (assistant replies, tool calls/results, etc.) **until the next real user message**.
- You configure two knobs:
  - `context_limit`: the maximum number of **real user turns** allowed in the raw history before we summarize.
  - `keep_last_n_turns`: how many of the most recent **turns** to keep verbatim when we do summarize.
    - Invariant: `keep_last_n_turns <= context_limit`.

- When the number of **real** user turns exceeds `context_limit`, the session:

  1. **Summarizes** everything **before** the earliest of the last `keep_last_n_turns` turn starts,
  2. Injects a **synthetic user→assistant pair** at the top of the kept region:
     - `user`: `"Summarize the conversation we had so far."` (shadow prompt)
     - `assistant`: `{generated summary}`
  3. **Keeps** the last `keep_last_n_turns` turns **verbatim**.

This guarantees the last `keep_last_n_turns` turns are preserved exactly as they occurred, while all earlier content is compressed into the two synthetic messages.

```
session = SummarizingSession(
    keep_last_n_turns=2,
    context_limit=4,
    summarizer=LLMSummarizer(client)
)
```

```
# Example flow await session.add_items([{"role": "user", "content": "Hi, my router won await session.add_items([{"role": "assistant", "content": "Let's check y await session.add_items([{"role": "user", "content": "Firmware v1.0.3; await session.add_items([{"role": "assistant", "content": "Try a factory await session.add_items([{"role": "user", "content": "Reset done; error await session.add_items([{"role": "assistant", "content": "Try to insta await session.add_items([{"role": "user", "content": "I tried but I got await session.add_items([{"role": "assistant", "content": "Can you plea await session.add_items([{"role": "user", "content": "It says 404 not fo await session.add_items([{"role": "assistant", "content": "Are you conne # At this point, with context_limit=4, everything *before* the earliest # is summarized into a synthetic pair, and the last 2 turns remain verba
```

```
history = await session.get_items()
# Pass `history` into your agent runner / responses call as the conversa
```

```
history
```

```
[{'role': 'user', 'content': 'Summarize the conversation we had so
 {'role': 'assistant',
 'content': '• Product & Environment:\n - Router with Firmware v1
{'role': 'user', 'content': 'I tried but I got another error now.' {'role':
'assistant', 'content': 'Can you please provide me with the error
code?'}, {'role': 'user', 'content': 'It says 404 not found when I try
to access the page.' {'role': 'assistant', 'content': 'Are you
connected to the interne
```

```
print(history[1]['content'])
```

```
• Product & Environment:
- Router with Firmware v1.0.3, Windows 10, based in the US.

• Reported Issue:
- Router fails to connect.

• Steps Tried & Results:
- Checked FAQs: No resolution.
- Checked firmware version: v1.0.3, problem persists.
- Factory reset: Resulted in error 42.

• Identifiers:
- Premium customer (no specific identifier provided).

• Timeline Milestones:
- Initial troubleshooting via FAQs.
- Firmware check (before factory reset).
- Factory reset → Error 42.

• Tool Performance Insights:
- Firmware version check successful.
- Factory reset resulted in new error (42).

• Current Status & Blockers:
```

You can use the get_items_with_metadata method to get the full history of the session including the metadata for debugging and analysis purposes.

```
full_history = await session.get_items_with_metadata()
```

```
full_history
```

```
[{'message': {'role': 'user',
    'content': 'Summarize the conversation we had so far.'},
 'metadata': {'synthetic': True,
    'kind': 'history_summary_prompt',
    'summary_for_turns': '< all before idx 6 >'}},
 {'message': {'role': 'assistant',
    'content': '**Product & Environment:**\n- Device: Router\n- OS: W
 'metadata': {'synthetic': True,
    'kind': 'history_summary',
    'summary_for_turns': '< all before idx 6 >'}},
 {'message': {'role': 'user',
    'content': 'I tried but I got another error now.'},
 'metadata': {'synthetic': False}},
{'message': {'content': 'I still have a problem with my router.',
    'role': 'user'},
 'metadata': {'synthetic': False}},
{'message': {'content': [], 'role': 'user'},
'metadata': {'id': 'rs_68ba192de700819dbed28ad768a9c48205277fe332
    'summary': [],
    'type': 'reasoning',
    'synthetic': False}},
 {'message': {'content': [{'annotations': [],
      'text': 'Sorry you're still stuck. What is the exact error cod
      'type': 'output_text',
```

```
print(history[1]['content'])
```

```
**Product & Environment:** - Device: Router - OS: Windows 10 - Firmware: v1.0.3

**Reported Issue:**
- Router fails to connect to the internet, now showing error 42.

**Steps Tried & Results:**
- Checked FAQs: No resolution.
- Firmware version checked: v1.0.3.
- Factory reset performed: Resulted in error 42.

**Identifiers:**
- UNVERIFIED

**Timeline Milestones:**
```

```
- User attempted FAQ troubleshooting.
- Firmware checked after initial advice.
- Factory reset led to error 42.

**Tool Performance Insights:**
- FAQs and basic reset process did not resolve the issue.
```

Notes & design choices

- **Turn boundary preserved at the "fresh" side**: the `keep_last_n_turns` **user turns** remain verbatim; everything older is compressed.
- **Two-message summary block**: easy for downstream tooling to detect or display (`metadata.synthetic == True`).
- **Async + lock discipline**: we **release the lock** while the (potentially slow) summarization runs; then re-check the condition before applying the summary to avoid racey merges.
- **Idempotent behavior**: if more messages arrive during summarization, the post-await recheck prevents stale rewrites.

Evals

Ultimately, **evals is all you need** for context engineering too. The key question to ask is: *howdoweknowthemodelisn't"losingcontext"or"confusing context"?*

While a full cookbook around memory could stand on its own in the future, here are some lightweight evaluation harness ideas to start with:

- **Baseline & Deltas:** Continue running your core eval sets and compare before/after experiments to measure memory improvements.
- **LLM-as-Judge:** Use a model with a carefully designed grader prompt to evaluate summarization quality. Focus on whether it captures the most important details in the correct format.
- **Transcript Replay:** Re-run long conversations and measure next-turn accuracy with and without context trimming. Metrics could include exact match on entities/IDs and rubric-based scoring on reasoning quality.

- **Error Regression Tracking:** Watch for common failure modes—unanswered questions, dropped constraints, or unnecessary/repeated tool calls.
- **Token Pressure Checks:** Flag cases where token limits force dropping protected context. Log before/after token counts to detect when critical details are being pruned.
