缺失的内存层次结构：大语言模型上下文窗口的按需分页

Tony Mason 不列颠哥
伦比亚大学 加拿大 佐治亚理
工学院 美国
fsgeek@cs.ubc.ca
fsgeek@gatech.edu
fsgeek@wamason.com

arXiv:2603.09023v1 [cs.OS] 9 Mar 2026

摘要

大语言模型的上下文窗口不是内存。它是 L1 缓存——一种小而快、代价高昂的资源，而该领域却将其视为整个内存系统。没有 L2，没有虚拟内存，也没有分页。每一个工具定义、每一条系统提示，以及每一个过时的工具结果，都会在会话的整个生命周期内占用上下文。结果是可量化的：在 857 个生产会话和 44.5 亿个有效输入 token 中，21.8% 属于结构性浪费。

我们提出 Pichay，一种面向大语言模型上下文窗口的按需分页系统。它以位于客户端与推理 API 之间的透明代理形式实现，介入消息流，以驱逐过时内容，在模型重新请求已驱逐材料时检测缺页，并固定由缺页历史识别出的工作集页。在涵盖 140 万次模拟驱逐的离线回放中，故障率为 0.0254%。在持续 681 轮的线上生产部署中，该系统将上下文消耗最多降低 93%（5,038KB → 339KB）；在极端持续压力下，系统仍可运行，但表现出预期的抖动病态，即反复将已驱逐内容缺页调入。

关键观察在于，该领域面临的问题——上下文限制、注意力退化、成本扩展，以及跨会话状态丢失——都是披着不同外衣的虚拟内存问题。解决方案已经存在：工作集理论 [3],按需分页、缺页驱动的替换策略，以及包含多个由驱逐管理的层级的内存层次结构。我们描述了面向大语言模型系统的完整内存层次结构架构（从 L1 到持久化存储），报告了已投入生产使用的前三个层级（L1 淘汰、L2 缺页驱动的固定、L3 模型发起的会话压缩），并将跨会话记忆确定为尚待探索的前沿。

CCS 概念：• **软件及其工程** → **分配／释放策略**；**虚拟内存**；•**计算方法** → 自然语言处理。

关键词：上下文窗口、虚拟内存、按需分页、大语言模型、智能体人工智能、工作集、驱逐策略

ACM 参考格式：
Tony Mason。2017年。《缺失的内存层次结构：大语言模型上下文窗口的按需分页》。载于*ACM SIGOPS* 第 *31* 届操作系统原理研讨会论文集（*SOSP '17*）。ACM，美国纽约州纽约，15 页。

SOSP '17，中国上海
2017年。

1 引言

1961 年，Atlas 计算机引入了按需分页：程序能够访问比物理内存实际容量更大的内存，而硬件会根据需要，从后备存储透明地加载页面 [8]。在 Atlas 之前，程序员需要手动管理覆盖——显式地将代码段换入或换出固定的地址空间。从覆盖到虚拟内存的转变，是系统架构领域的基础性进步之一。

大语言模型上下文窗口正处于覆盖时代。每个智能体人工智能工具——Claude Code、Cursor、GitHub Copilot、Windsurf——都会在每次 API 调用时手动组装上下文：工具定义、系统提示词以及完整的消息历史。这里没有驱逐策略，没有按需加载，也没有工作集估算。默认行为是不断累积，直到上下文限制迫使系统陷入危机。

本文提出两项贡献。首先，我们提供了实证证据，表明上下文管理本质上是内存管理问题。我们对 857 个生产环境会话（54,170 次 API 调用、44.5 亿个有效输入令牌）进行了插桩分析，发现 21.8% 的词元属于结构性浪费，来源于三个方面：未使用的工具模式（11.0%）、重复内容（2.2%）以及过时的工具结果（8.7%），后者以中位数 84.4× 倍的放大比例被重新处理。这些并非实现缺陷，而是在缺乏操作系统五十年前为类似问题发展出的抽象机制时，对有限资源（上下文窗口）进行管理所不可避免的结果。

其次，我们构建了这个系统。Pichay 是一种面向大语言模型上下文窗口的按需分页系统，实现为客户端与推理 API 之间的透明代理。它能够区分

垃圾回收（移除无法重新请求的短暂工具输出）与分页（驱逐可寻址、且可通过缺页重新载入的内容）。当模型重新请求已驱逐的内容时，它会检测到缺页。它会固定由缺页历史识别出的工作集页面。它生成的驱逐摘要——旨在作为节省空间的标记——可充当模型无需指示便能理解的检索句柄："[已分页移出：Read file.py（8,192 字节数）。如有需要，请重新读取。]"

在生产部署中，系统将会话的空闲上下文比例从 7% 提升至 43%——这意味着会话不再濒临上下文耗尽，而是拥有了充足的剩余容量。在一场持续 681 轮的会话中，即使驱逐率高达 97%，系统仍能持续运行，不过这个极端案例也清楚展现了操作系统研究中众所周知的抖动问题 [3]：当工作集超过驻留集时，系统花在缺页上的精力会超过有效工作。

更深层的观察是：上下文窗口并非内存，而是 L1 缓存。可没人着手构建其余层级。面对上下文限制，该领域的应对方式是扩大 L1——推出 100 万词元、1000 万词元的窗口。这相当于不断增加物理内存，而不是发明虚拟内存。虽然可行，但无法扩展，也错过了架构层面的关键洞见：由小而快、成本高的存储与大而慢、成本低的存储构成的分层结构，通过驱逐策略和缺页机制连接起来，无论单一层级扩大到多大，都胜过只用单一层级。

我们描述了完整的内存层次结构：L1（生成窗口）、L2（工作集，按需分页并固定）、L3（会话历史，经过压缩并声明损失）以及 L4（跨会话持久化内存，已建立索引并可检索）。前两个层级已经部署并完成评估。L3 已实现——模型可以通过协作清理标签将对话轮次压缩为结果摘要——但尚未在大规模场景下进行评估。L4 已完成设计，其机制也已确定。

主要贡献：

1. 对生产环境中的智能体人工智能语料库进行了大规模实证分析，刻画上下文窗口的利用情况，并通过测量得出分类体系，确定有 21.8% 的词元属于结构性浪费。2. Pichay：一种面向大语言模型上下文窗口的按需分页系统，其缺页故障率经过实证测量（离线环境为 0.0254%，并已部署到生产环境）。3. 故障驱动固定：一种能够从自身错误中学习的页面置换策略——一次缺页故障即可在整个会话期间固定某个页，从而减少稳态工作负载中的重复缺页故障。4. 通过两条旁路信道（幻影工具和清理标签）实现协作式内存管理：相关机制使模型能够主动释放冷页面、请求缓存缺页故障，并压缩对话历史

提出了一个硬件内存层次结构中不存在的新设计空间，因为其中的应用程序是不协作的。5. 这一架构性观察指出，大语言模型上下文管理在结构上（而不仅仅是比喻意义上）映射到虚拟内存，并且完整的操作系统内存层次结构——从缓存置换策略到工作集理论——都可以直接应用。

6. 为大语言模型系统设计了四级内存层次结构，其中前三级已经部署（L1 淘汰、L2 固定、L3 协同压缩），前两级已完成评估。

2 背景

2.1 AgenticAI 架构

现代 AI 编码助手以智能体系统的方式运行：模型会接收一组工具定义（描述可用操作的 JSON 模式）、一条系统提示词（包含身份、指令和行为约束），以及不断增长的消息历史（由交替出现的用户与助手轮次组成，其中包括工具调用及其结果）。

每次 API 调用时，客户端应用程序都会组装完整的上下文：系统提示词、工具定义和完整的消息历史。推理提供方会对该上下文进行分词，计算所有词元的注意力，并生成响应。随后，响应会追加到消息历史中，循环往复。

关键特性：每次 *API* 调用时，都会从头重新组装上下文。推理层在调用之间没有持久状态。整个对话——包括未发生变化的工具定义，以及之后再也不会引用的结果——都会在每一轮中被发送、分词并计算注意力。

2.2 作为物理内存的上下文窗口

上下文窗口与物理内存之间的类比是结构性的，而不仅仅是修辞性的（表 1）。

表 1. 上下文窗口管理的虚拟内存类比。

操作系统概念	上下文窗口对应项
物理内存	上下文窗口（20 万词元）
虚拟内存	持久状态（磁盘、数据库）
页表	检索句柄（通用唯一标识符、锚点）
缺页	模型重新请求已驱逐的内容
MMU	客户端与 API 之间的代理层
工作集	当前相关的上下文子集
按需分页	首次使用时加载工具定义
抖动	驱逐/缺页故障周期耗费的时间超过有效工作
固定	缺页历史可防止再次驱逐

早期计算机系统通过手动覆盖来管理有限的物理内存：程序员会明确地将代码段换入和换出固定的地址空间。从 Atlas [8], 、Multics 到 Unix，虚拟内存的发展通过硬件页表、按需分页和基于工作集的淘汰策略 [3,4] 实现了这一管理过程的自动化。

上下文窗口仍处于覆盖时代。应用程序需要手动管理可容纳的内容，系统层面没有提供驱逐、按需加载或工作集估算支持。

2.3 内存层次结构

面对上下文压力，这一领域的应对方式是增大上下文窗口大小——从 4K 增至 32K、128K、1M 词元乃至更大。这相当于不断制造物理内存更大的机器，而不是发明虚拟内存。虽然这种做法有效，却无法扩展：注意力开销随上下文长度呈二次方增长，而长会话中的 $O(n^2)$ 成本激增意味着，即使是百万词元窗口也会被填满并出现性能下降 [17]。

另一种方案着眼于架构：构建由多个层级组成的受管理层级结构，每一层都比上一层更大、更慢、成本更低，并通过驱逐策略和缺页机制连接起来（表 2）。

表 2. 大语言模型系统的内存层次结构。

层级	内容	驱逐	缺页
L1	生成窗口	—	—
L2	工作集（固定）	基于压力	重新读取
L3	会话历史	基于时长	摘要扩展
L4	跨会话记忆	最近最少使用 / 相关性	检索查询
存储	全量语料	None	搜索 + 摄取

L1 是活跃的生成窗口，即模型在当前 API 调用中关注的词元。容量小、速度快，但每个词元的成本高。这就是大家所说的“上下文窗口”。

L2 是工作集：模型正在积极使用、但不必在每一轮都占据 L1 的内容。通过故障驱动固定进行管理（第 3.5 节）。从 L1 驱逐后又立即被请求的页会提升至 L2，并保持驻留，直到上下文压力迫使其降级。

L3 是会话历史：较早的对话轮次、已完成的工具交互、已取代计划。通过声明损失进行压缩，形成结构化摘要。需要时可从客户端的后备存储中将摘要还原为原始内容，以处理缺页。

L4 是跨会话持久记忆：编写的压缩内容（张量）、活动记录、语义索引。即使会话结束也会保留。通过图遍历或相似性搜索进行检索。

存储 是全量语料：每一份对话转录、每一条工具输出、每一份文档。它们经过归档和索引，只有在需要时才会驻留。

内容会根据访问模式在各层级之间迁移。热路径保留在 L1 中。工作集内容会在 L1 和 L2 之间按需分页。冷历史记录会压缩到 L3。跨会话知识会持久保存在 L4。上下文限制并未消失，而是变成了 L1 缓存大小。可寻址总量不受限制。系统性能取决于各层级的命中率，而非仅由 L1 容量决定。

本文实现并评估了 L1 管理（驱逐和垃圾回收）以及 L2 管理（故障驱动固定）。L3（滚动式对话压缩）通过协作清理标签实现（第 3.9 节），但尚未进行大规模评估。我们介绍了 L4（持久化记忆存储）的接口。

2.4 相关工作

上下文压缩。LLMLingua [5] 及相关方法通过移除预测为低信息量的词元来压缩提示词。这些方法是有损的、依赖模型的内容处理操作。我们的干预措施是结构性的：我们移除整个类别的浪费（未使用的工具模式、重复内容、无效结果），而不对剩余内容进行有损压缩。

智能体上下文剪枝。 SWE-Pruner [15] 训练了一个 0.6B 参数的神经网络筛选器，用于 SWE-bench 智能体中面向任务的上下文剪枝，实现了 23–54% 的词元缩减。Lindenbauer 等人 [9] 表明，只需屏蔽观测，就能将智能体成本减半，同时保持与大语言模型摘要相当的任务解决率。ACON [6] 提供了统一的历史记录与观测压缩框架，将峰值用量降低了 26–54%。

这些方法将问题视为提示词优化或压缩。关键区别在于：它们针对固定任务分布上的基准测试得分进行优化。我们衡量生产环境浪费概况，并实施基于实测故障率的驱逐策略。它们的思路是“让提示词更短”。我们的思路是“管理工作集”。工作集管理可以与其他干预措施结合，并提供可衡量的故障率。

二次成本分析。 近期工作已量化长智能体会话中的 $O(n^2)$ 成本激增，其中缓存读取主导 [17]。每次 API 调用都会重新处理完整上下文；随着会话增长，累计成本随会话长度呈二次增长。我们的放大因子测量（第 5.1 节）在生产规模上证实了这一点，而我们的干预措施直接针对该问题。

提示词缓存。 Anthropic 和 OpenAI 提供提示词缓存，可避免对静态前缀进行重复计算。在我们的语料库中，93.5% 的输入令牌是缓存读取——这说明缓存正在发挥作用。但缓存的词元仍会占用上下文窗口，并且每生成一个输出词元都需要进行注意力计算

。提示词缓存降低了输入处理开销，但不会降低注意力开销或内存压力。

上下文工程。 新兴的上下文工程学科 [2, 10, 16] 认为，管理进入上下文的内容与模型本身同样重要。Mei 等人[10] 对该领域进行了全面综述，涵盖检索、处理与管理。Anthropic 自己的工程团队描述了实用模式 [12]。我们的工作提供了一种具体机制（带有驱逐策略的代理层插入），以及该领域目前所缺乏的经验性生产数据。

提示级内存管理。 MemGPT [11] 率先提出了操作系统虚拟内存类比：一个在达到容量时进行递归摘要的先进先出消息队列，外加归档/r
ecall 数据库可通过工具调用访问。模型可以保存信息，但无法显式释放它；驱逐由容量触发（在 100% 时刷新），而非由成本驱动。上下文内存虚拟化 [13] 将会话历史建模为有向无环图，并通过结构无损修剪去除机械性冗余（原始工具输出、Base64、元数据），同时逐字保留对话内容——在工具密集型会话中最多可减少 86%。Focus [14] 赋予智能体对压缩的自主权：大语言模型将所学内容整合到一个持久知识块中，并修剪原始历史，其灵感来自黏菌探索。SideQuest [7] 通过一条微调的并行推理线程教会模型从 KV缓存中驱逐过时的工具输出，使峰值内存降低 56–65%——但驱逐不可逆，且需要推理引擎修改。

这些系统分别解决了架构问题（MemGPT 的虚拟机类比）、结构问题（CMV 的无损裁剪）、自主性问题（Focus 的模型引导压缩）或推理问题（SideQuest 的 KV缓存驱逐）。没有任何一种系统推导出这样的经济模型：提示词词元的保留成本呈二次增长，而缺页成本呈线性增长。也没有任何一种系统提供依据该成本模型设计的协作式召回/释放原语。没有任何一种系统报告生产会话中的实测故障率。

定位。既有工作要么解决成本问题（二次分析），要么解决内容问题（压缩、剪枝、检索增强生成），却没有解决结构问题：上下文窗口是未受管理的物理内存。我们的贡献是提出一种系统抽象——工作集、驱逐策略、故障率、放大因子——并提供其在生产数据上有效的实证证据。

3 系统设计

Pichay 是一个透明的 HTTP 代理，介入智能体 AI 客户端与推理 API 之间。每次收到请求时，它都会接收客户端组装的完整消息数组，应用上下文管理策略，并将修改后的请求转发给推理提供方。客户端不会察觉到这一介入。

3.1 架构

该代理运行在 Messages API 层。它接收包含系统提示词、工具定义和消息历史的 JSON 请求，并原样返回推理提供方的响应。其介入点是消息数组：代理可以在转发前检查、测量、修改或替换任意消息。

这种设计具有三个特点：(1) 无需更改客户端、模型或推理 API；(2) 客户端保留完整且未经修改的对话历史，提供后备存储，以便在内容被逐出后将其重新调入内存；(3) 代理在每一轮都能看到完整请求，从而无需持久连接即可进行有状态管理。

3.2 垃圾回收与分页

并非所有工具结果都相同。我们将其分为两类：

垃圾回收适用于短暂工具输出——Bash 命令结果、搜索输出、目录列表——这些内容没有稳定标识。一旦被使用，就无法再通过有意义的方式重新请求。移除它们属于对无用内容的回收。

分页适用于可寻址内容——文件读取、计划文档、规范——这些内容具有稳定标识（文件路径）。移除这些内容会带来缺页风险：模型可能再次需要这些内容，因此必须重新读取文件。

这一区别对故障率的计算很重要。只有分页（Read 驱逐）会导致缺页故障，垃圾回收则不会。将两者混为一谈会扩大驱逐分母，从而低估表面上的故障率。

3.3 驱逐策略

当前策略是按用户轮次先进先出：早于 τ 个用户轮次且大小超过 $s_{\min}$ 字节数的工具结果可作为驱逐对象（所有实验中均为 $\tau = 4$、$s_{\min} = 500$）。这一策略刻意保持简单。研究问题在于，最简策略在需要更复杂的方法之前能适用到什么程度。

被驱逐的内容会替换为一个检索句柄：

> [已分页移出：Read /path/to/file.py（8,192 字节，187 行）。如果需要其内容，请重新读取该文件。]

该句柄有三个作用：(1) 标明移除了什么；(2) 说明如何恢复内容；(3) 与原内容相比，占用的上下文空间大幅减少。

3.4 缺页检测

当模型发出与已驱逐条目匹配的工具调用（工具名称和参数相同）时，代理会检测到缺页：模型请求的是它之前拥有的内容

但这些内容已因驱逐而丢失。已驱逐条目通过由工具名称和输入参数派生的键进行索引（例如 Read 操作中的文件路径）。

缺页是可观测事件。它们衡量驱逐的代价：每次缺页都意味着一次白费的工具调用往返；如果内容一直驻留在内存中，这种往返本不会发生。按照推理时定价，缺页会产生直接的金钱成本。

3.5 故障驱动固定

对先进先出最简单的改进：如果驱逐某个页导致了缺页，就不要再次驱逐它。每个文件只发生一次缺页，此后在整个会话期间永久固定。算法如下：

1. 驱逐可寻址内容时：记录文件路径和内容哈希。
2. 发生缺页时：将被驱逐内容的哈希记录在故障历史表中。3. 下次尝试驱逐同一路径时：如果当前内容哈希与故障历史表中的条目匹配，则固定该页——永久跳过驱逐。4. 对内容不同的固定路径进行新的读取时（文件已编辑）：取消固定。旧版本已过时；新版本开始新的缺页周期。

内容哈希比较可防止错误固定：如果文件在驱逐后、重新读取前发生了变化，那么此次驱逐就是正确的（已移除过时数据）。只有当模型确实需要被移除的内容时，才会进行固定。编辑时取消固定的规则则适用于编码会话中常见的编辑/审查周期。

3.6 作为锚点的检索句柄

驱逐摘要原本是作为一种节省空间的标记而设计的。实际使用中，它起到了后绑定检索句柄的作用——一种仅存储最少元数据、并可按需解析为完整内容的锚点。

该句柄解析为当前内容，而非最初被驱逐的内容。文件若在驱逐后经过编辑，那么在发生缺页时，会以其新状态载入。这一时间属性是一项优势：模型始终获取最新版本，也就不可能出现过时的缓存内容。

行为证据：一个新的模型实例恢复了包含已换出的内容的会话后，主动表示："让我重新读取所需的文件，因为有些内容已被换出。"模型识别出了这些句柄，理解缺少了哪些内容，并选择先载入内容再采取行动。句柄的格式本身就承载了语义——无需额外指令。

3.7 协作式内存管理

在操作系统虚拟内存中，应用程序并不配合。它从不主动释放页。由于 CPU 不会告知接下来需要什么，操作系统必须根据访问模式推断工作集。正因存在这种根本性的不对称，才有了数十年来的置换算法——先进先出、最近最少使用、时钟置换算法、工作集算法。

大语言模型打破了这一假设。模型有动力 与内存管理器协作：更整洁的上下文意味着更好的注意力、更高的输出质量，以及更长的会话寿命。处理器希望帮助管理自己的缓存。这改变了设计空间。

Pichay 利用这一机制实现了幻影工具——由代理注入、框架完全看不到的工具定义。模型可以调用它们；代理会在框架收到流式响应之前拦截其中的调用，在内部处理，并在下一轮注入一致的工具结果。框架并不知道这个侧信道。

定义了两个幻影工具：

内存_释放(paths) 模型表示不再需要特定文件。代理会将这些文件标记为立即驱逐，绕过老化阈值。这就是引用位——由处理器自愿提供。

内存_缺页(paths) 模型请求从代理的后备存储中恢复已驱逐的内容。代理无需经过文件系统往返，即可通过驱逐缓存解决缺页——这比调用真正的 Read 工具更快、更省成本。

第二个协作通道是清理标签——嵌入模型输出文本中的结构化指令，代理会在转发给框架之前解析这些指令。幻影工具是代理到模型的通道（代理提供功能），而清理标签则是模型到代理的通道（模型发起管理操作）。定义了四种操作：

丢弃：块:ID 立即驱逐特定块。
概述：块:ID "text" 将一个块替换为压-缩摘要——由模型而非系统撰写的有损压缩。

锚定：块:ID 固定一个块，使其免遭驱逐。
折叠：轮次 N-M "text" 将一段对话轮次压缩为单个合成摘要块，并移除所有中间内容。

折叠操作尤为重要：它使模型能够自行发起 L3 压缩（第 3.9 节），依据的是模型对哪些对话只是脚手架、哪些产生了持久结果的理解。

关键洞见：在硬件内存层次结构中，应用程序是对抗性的或漠不关心的。整个置换算法文献都假设不合作。协作式按需分页——即处理器自愿释放冷页面并显式请求缺页故障——是设计空间中的一个新点；其得以实现，是因为大语言模型在上下文压力下会出现输出质量下降，并且能够学会合作可以提升自身性能。这两个通道（幻影工具和清理标签）提供了协作协议的两个方向：代理将内存状态告知模型，而模型则根据其对何为冷内容的认知评估，指示代理采取行动。

3.8 分级压力区

驱逐策略必须响应上下文压力，但二元阈值（驱逐/不驱逐）过于粗略。Pichay 根据令牌消耗定义了四个压力区：

正常 (< 60K 个词元**)**：不进行干预。代理进行观察并记录日志。

通知 (60K–100K**)**：代理将内存压力信息注入模型上下文，包括当前填充率、最大的五个驻留块以及可用的清理操作。模型可以配合采取行动，也可以忽略通知。这相当于桌面操作系统中的“内存不足”通知，只不过采用了分级方式。

强制 (100K–120K**)**：代理开始使用先进先出年龄策略自动逐出。模型会收到通知，但不会被征询意见。

激进型 (≥ 120K**)**：紧急驱逐，采用宽松阈值。优先确保上下文存续，而非保留工作集。

建议区体现了协作式创新。系统不会像操作系统那样静默驱逐，也不会像当前的智能体人工智能方案那样在达到容量上限时崩溃，而是向模型提供充足信息，帮助其做出明智的清理决策。选择 60K 这一阈值，是为了在非自愿驱逐发生前留出约 40K 词元的缓冲空间——足以让模型完成连贯的思考，并在失去对流程的控制权之前发出清理标签。

3.9 L3：滚动式对话压缩

代理管理工具结果（L1/L2）。非工具内容——对话历史、规划对话、环境了解读取——则由折叠操作管理（第 3.7 节）。这是内存层次结构中的 L3：会话历史经过压缩，并明确声明损失，形成结构化摘要。

折叠操作 collapse：turns N-M "summary" 会将一个连续轮次范围内的所有块替换为一个包含模型撰写摘要的合成块。原始内容不会被存储——这是有意采用的有损压缩。摘要关注的是结果 （做出了什么决定、构建了什么、哪些事项失败了），而非过程 （读取了哪些文件、经历了哪些中间步骤）。

块状态通过原子检查点机制在会话重启后持久保存。每次处理清理标签时，代理都会使用原子写入（临时文件 + 重命名）将块元数据（ID、内容哈希、大小、轮次、角色、状态、摘要）序列化到 JSON 检查点文件中。创建会话时，系统会加载检查点，并从持久化状态恢复块跟踪。重新处理消息数组时，原始内容会按需重新填充。

检查点仅包含元数据——内容不会持久化（客户端的消息数组就是后备存储）。这样可以使检查点文件保持较小的体积（以千字节计，而非兆字节），并避免维护两份消息内容所带来的一致性风险。

4 测量

4.1 语料库

我们对 Anthropic 的人工智能编程助手 Claude Code 进行监测，覆盖一名重度用户在约四个月内（2025年11月至2026年3月）的会话。语料库包含 857 个会话，涵盖 15 个软件项目，数据来自两个来源：本地 WSL 环境和 Ubuntu 虚拟机（从之前使用的机器迁移而来）。

会话按类型分类：

- **主会话**（59 个）：面向用户的对话，用户直接与助手交互。
- **子代理会话**（567）：主会话启动的委派任务，用于并行处理子任务。
- **压缩会话**（154）：当主会话接近上下文限制时触发的上下文压缩延续（客户端会自动概述对话历史，使其保持在上下文窗口范围内）。
- **提示词建议会话**（21）：用于生成提示词建议的短会话。

该语料库包含 54,170 次 API 调用和 44.5 亿个有效输入令牌（输入令牌 + 缓存创建 +缓存读取令牌）。令牌使用量取自嵌入会话记录中的 API 响应元数据。

由于实时语料库仍在不断变化，本文报告的是一个冻结队列（857 个会话），而非持续变动的实时计数。我们提供了一个重新计数脚本（tools/corpus_counts.py），以明确计数边界和去重假设。除非另有说明，本稿中的所有语料库表格和图示均采用固定计数流程（按内容去重，最小_大小=10000）统计冻结队列，以确保结果在不同修订版本之间保持稳定。

偏差声明。这是一份来自重度用户的单用户语料库，该用户在处理复杂的多文件工程任务时经常耗尽上下文窗口。我们认为，这类用户最能代表上下文管理至关重要的群体；此外，考虑到使用量呈帕累托分布，这类用户很可能占据整个用户群体词元消耗的主要份额。多用户验证被列为未来工作。

4.2 工具

我们在技术栈的不同层次开发了三种工具：

探针。 (probe.py)：一个流式 JSONL 分析器，用于读取 Claude Code 的原始会话记录。按类型（用户、助手、工具_结果、进度）对记录进行分类，衡量内容大小、跟踪工具使用情况，并计算每会话指标，包括放大因子和工具开销比。无需 API 调用；直接处理现有会话文件。

代理。 (proxy.py)：部署在 Claude Code 与 Anthropic API 之间的透明 HTTP 代理。它会拦截每个请求，捕获系统提示词、工具定义、消息数组和令牌使用量。还可选择在转发请求前应用干预措施（驱逐、裁剪）。所有决策都会记录到 JSONL 中，以便离线分析。这就是我们类比中的“内存管理单元”。

实验运行器。 (Pichay)：一个实验框架，用于协调不同处理条件（基线、裁剪、压缩+裁剪）下的配对运行，收集所有产物（代理日志、会话数据、Git 状态、测试结果），并计算比较指标。该框架能够自举：它由 Claude Code 通过自身代理构建而成，每一轮的日志都会用于下一轮分析。

4.3 处理条件

我们定义了三种应用于代理层的处理条件：

1. **基线**：代理处于观测模式。记录所有流量，不做修改。 2. **裁剪**：在转发每个请求之前，对其应用工具定义存根化和技能去重。 3. **压缩+裁剪**：将过期结果淘汰（分页）与裁剪相结合。

所有处理组使用相同的模型（Claude Opus）、相同的温度和相同的任务提示词。代理的介入对客户端和 API 都是透明的 —— 双方都不会察觉这些修改。

5 结果

分析分为两个阶段。第一阶段通过分析会话转录，使用全量 857 个会话的语料库，概述广泛的对话模式——包括工具开销比、放大因子和令牌核算。第二阶段则针对一个规模较小、由代理捕获的样本（5 个会话、99 次 API 调用），进行详细的 API 级浪费分解；该样本包含完整的请求负载。随后，语料库规模的推算（第 5.6 节）将两个阶段衔接起来：把第二阶段测得的换算常数应用于全量语料库，以估算整个集群中可处理的浪费。用于推算的换算常数（字节与令牌比率）根据一个包含 139 次 API 调用的更大代理样本估算得出。

5.1 阶段 1：对话层面的浪费

工具开销。 在全量语料中，79.4% 的对话字节数来自工具结果。助手文本占 12.7%，用户文本占 7.9%。两项独立测量得出的比例相近（78.2% 和 79.4%），这表明该比例是工作负载的稳定特征，而非抽样造成的假象。

工具类型集中度。 工具使用情况极为集中（图1）。Read 占所有工具输出字节的 75%（9,393 次调用，平均每个结果 7,935 字节）。Bash 占 13.3%（10,090 次调用）。其他所有工具各自的占比均不足 5%。这种集中度直接意味着：Read 工具读取的文件内容是上下文膨胀的主要来源。

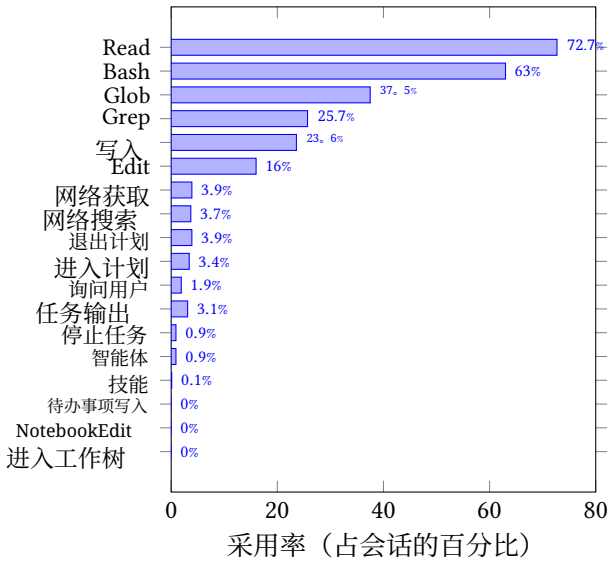

图1。801 个活跃会话中的工具采用率。中位数会话使用 18 种可用工具中的 3 种。7 种工具的采用率为零或接近零，但每次 API 调用仍会发送其完整模式（合计 ~24，500 字节）。

放大因子。每个工具结果都会从生成该结果的轮次起保留在上下文中，直到会话结束（或被客户端的内部压缩移除）。我们将放大因子定义为：

$$A = \frac{\sum_{r \in R} \text{size}(r) \times \text{turns\_survived}(r)}{\sum_{r \in R} \text{size}(r)}$$

其中，R 是工具结果集合，而 survived turns 是指该结果仍保留在上下文中的后续轮次数量。该指标衡量每个字节的工具输出平均会被重新处理多少次。

主会话：中位数 $A = 84.4\times$、P75 = 217.9×、P90 = 570.8×。子代理：中位数 $A = 12.8\times$（生命周期较短，累积较少）。放大程度随会话长度线性增长，比例约为 0.5，这证实了近期分析 [17] 所指出的二次成本结构。

令牌核算。该语料库消耗了 44.5 亿个有效输入令牌，缓存命中率为 93.5%，这意味着绝大多数输入处理都由缓存完成。然而，缓存中的令牌仍会占用上下文窗口，并且每生成一个输出令牌都需要进行注意力计算。每次 API 调用平均处理 82,061 个有效输入令牌，并生成 88 个输出令牌，输入输出比为 933:1。智能体编程是一种极度受输入限制的工作负载。

5.2 阶段 2：API 级别的浪费分类

使用该代理，我们将 5 个会话（99 次 API 调用，24.4 MB 请求数据）分解为四类可处理浪费（表 3）。

表 3. 浪费分类。百分比以 99 次 API 调用的请求字节数总量为基准。

类别	字节数	%	机制
失效工具输出	6,468,360	26.5	过时结果永远不会再次被引用
工具定义存根	4,924,950	20.2	未使用工具的模式
静态重发	2,680,794	11.0	未更改的系统提示
技能三重化	700,582	2.9	列出的相同技能 3×
可寻址总量	**14,774,686**	**60.5**	

失效工具输出（***26.5%***）。早期轮次中的工具结果会一直保留在上下文中，即使之后再也没有引用它们。在定向阶段（会话的第 1 轮）读取的文件，在会话剩余时间约 90% 的过程中都会留存。每次后续 API 调用都会重新处理这些内容，消耗注意力，却无助于当前任务。

工具定义模式（***20.2%***）。Claude Code 每次 API 调用都会发送 18 个工具定义，总计 63,088 字节数。会话中位数使用 3 个工具。其余 15 个工具模式（~52,500 字节数）每次调用都会被发送、分词并参与注意力计算，却从未被调用。

静态系统内容（***11.0%***）。系统提示词、CLAUDE.md 指令和记忆文件会在每次 API 调用时原样重新发送。虽然提示词缓存避免了前向传播成本，但这些词元仍占据上下文窗口，并参与注意力计算。

技能三重化（***2.9%***）。技能列表——用于描述可用的斜杠命令——会以三种不同前缀（基础、example-skills:base、document-skills:base）注入消息中。简单去重即可移除三分之二的条目。

5.3 干预措施

所有干预措施都在客户端应用程序与推理 API 之间的代理层中运行。无需对模型、客户端或 API 进行任何更改。

工具定义存根化。 未使用的工具定义会被替换为最小存根：

```
{"name": "NotebookEdit",
 "description": "...(第一行)...",
 "input_schema": {"type":"object","properties":{}}}
```

完整模式（~3，每个505字节）会被替换为桩（~80 字节）。首次调用桩工具时，完整定义会从存储的副本中恢复。该干预是会话级的：一旦某个工具被使用，其模式便会保持为该会话恢复。每次请求节省：$(18 - k) \times 3{,}425$字节，其中 k 是迄今已使用的工具数量。

内容去重。 技能条目经解析后，按基础名称分组并去重（保留首次出现的条目）。每个请求可节省 7,453 字节数。静态系统提示词组件会通过内容哈希跨轮次跟踪；相同的组件会被记录为前缀缓存候选。目前对静态组件仅进行测量——实际剥离需要支持缓存感知的 API。

过期结果淘汰（分页）。 超过一定时间（距离对话末尾已有 4 个用户轮次）且大小超过最低阈值（500 字节数）的工具结果会被淘汰。错误结果永不淘汰（模型需要它们进行调试）。被淘汰的内容会替换为摘要：

```
[已换出：Read /path/to/file.py（12,450 字节数，
287 行）。需要时重新读取。]
```

摘要会保留工具名称、关键参数和原始大小（~200字节数，无论原始大小是多少）。如果模型使用相同参数再次调用同一工具，就构成一次缺页——这说明淘汰有误，内容仍然需要。

5.4 淘汰安全性

我们通过离线回放验证驱逐安全性：将 29 个由代理捕获的会话通过分页器进行回放，在不进行 API 调用的情况下模拟驱逐决策。

表 4. 离线回放的淘汰安全性结果。

模拟驱逐总次数	1,393,000
检测到的缺页	354
故障率	**0.0254%**
已淘汰内容	8.49 GB

0.0254% 的故障率表明驱逐策略合理：超过 4 个用户轮次的内容几乎再也不会被需要。354 次缺页故障代表这些内容按照年龄标准确实已经失效，但恰好与后续请求相匹配。故障率为零则意味着驱逐过于保守；出现一些缺页故障是预期之内且可以接受的。

5.5 实时处理比较

我们在三种处理条件下运行一项标准化任务（表5）。

压缩+裁剪处理组可将有效输入令牌减少 37.1%。缓存读取减少 59.6%。在这次受控运行中，任务在所有条件下均能正确完成。

5.6 语料库规模预测

我们使用来自代理的测量常数，在完整的857个会话语料库中验证小样本发现

表5. 标准化任务上的处理组比较。

指标	基线	裁剪	压缩+修剪
API 调用	3	4	3
有效输入令牌	114,222	88,421	71,816
缓存读取令牌	79,712	38,639	32,228
任务已正确完成	Yes	Yes	Yes
词元缩减	—	22.6%	**37.1%**

会话以及从原始会话记录中提取的工具使用模式。

每个有效输入词元对应 4.15 字节的字节数与词元转换比率，是根据 139 次经代理捕获的 API 调用测得的。

表 6. 语料库规模词元节省预测。

干预措施	节省词元	占输入的百分比
工具存根裁剪	487.5M	11.0%
技能去重	95.8M	2.2%
静态重发	387.0M	8.7%
可寻址总量	**970.4M**	**21.8%**

平均每次 API 调用可节省 17,913 个词元。在整个语料库中，这意味着令牌间注意力对减少了 850 亿对（17,913 个上下文词元 × 88 输出词元 ×54,170 次调用）。

5.7 生产部署

我们以紧凑模式部署 Pichay，将其作为自身开发工作流的代理——作者每天都使用该系统完成软件工程任务。本节介绍两个具有代表性的会话，展示不同的运行模式。

会话 ***A***：稳态编码。 使用紧凑模式的标准编码会话（$\tau = 4$, $s_{\min} = 500$）：

表 7. 生产会话 A：稳态编码。

压缩前的可用上下文	7%
压缩后的可用上下文	43%
驱逐总数	15
垃圾回收	11
Read 驱逐（可分页）	4
缺页	1
故障率（仅 Read）	25%

该会话恢复了 36 个百分点的上下文容量——这意味着它避免了上下文即将耗尽，并保留了相当可观的剩余空间。在 15 次驱逐中，11 次属于垃圾回收（Bash/Grep/Glob 输出），另外 4 次是 Read 驱逐。唯一的一次缺页是会话早期对计划文件的读取，该文件在超过老化阈值后被驱逐。计划文件是整个会话都需要的参考资料——一个 FIFO 视为冷页的热页。这正是典型的工作集失效：驱逐策略衡量的是老化程度，而不是访问模式。

会话 ***B***：持续的多智能体协调。 涉及多个子智能体生成和跨项目文件分析的 A681-turn 会话：

表 8。 生产会话 B：持续 681 轮的会话。

轮次	681
驱逐（总计）	680
垃圾回收	74
Read 驱逐（可分页）	606
缺页	659
故障率（总驱逐数）	**97%（659/680）**
峰值压缩	5,038KB → 339KB
会话终止	API 速率限制

97% 的缺页率是一种病态现象，而非系统特性。系统几乎驱逐了所有内容，而模型又几乎重新请求了所有内容。出现了三种具体模式：

抖动循环。三个文件（3，438 + 1，570 + 1，592 字节）在第 163 至 170+ 轮之间循环被驱逐并发生缺页。工作集超出了驻留集——这是典型的抖动现象 [3]。

顺序扫描。 在规划阶段，模型遍历了三个代码库，反复循环读取相同的 7 个文件。规划所需的工作集大于老化阈值所允许保留在驻留集中的范围。

自我导致的膨胀。 5,038KB 的压缩前大小包含了先前驱逐所导致的重新读取。若没有驱逐-缺页循环，原始内容本会小得多。该代理把自身开销计为“节省的字节数”——这是抖动造成的指标伪象。

成本。 该会话因触及 API 速率限制而终止，而非上下文限制。在 659 次缺页故障（每次均为按推理计价的工具调用往返）下，抖动消耗速率额度的速度超过了有效工作。这表明缺页成本不仅是计算层面的，也是金钱层面的。

锚点句柄行为。 当一个全新的模型实例恢复包含换出内容的会话时，它表示：“让我重新读取需要的文件，因为有些文件已被换出，然后创建任务列表并开始实施。”模型识别出了检索句柄，理解缺少了什么，并选择先将内容调入，再采取行动——整个过程无需任何指示。句柄格式具有自描述性：模型仅凭摘要文本就能推断出恢复机制。

6 讨论

6.1 为什么会存在这种浪费

智能体人工智能工具中的只追加模式并非实现缺陷，而是由四个相互强化的因素造成的：

1. **训练数据**：模型从不断增长的对话中学习。没有训练示例展示如何智能地移除内容。2. **API设计**：Messages API 接受消息列表。列表的自然操作是追加。该 API 没有提供“此内容已过时”的机制。3. **框架默认设置**：每个编排框架默认都会追加内容。驱逐需要明确的工程实现。4. **不可见的成本**：词元消耗是在事后计费的。除非进行测量，否则上下文膨胀导致的质量下降是不可见的。没有背压信号。

6.2 颠倒的成本模型

在传统虚拟内存中，将一页保留在物理内存中是免费的。无论是否访问该页，都会分配随机存取存储器；唯一的成本是缺页——需要的页不在内存中时，磁盘输入/输出所带来的延迟。从先进先出到最近最少使用、时钟置换算法再到工作集，整个置换算法领域都在优化同一个目标：尽量减少缺页。

在大语言模型上下文管理中，成本模型恰好相反。上下文中保留的每个词元，每轮都会产生费用。一个包含 5,000 个词元的文件在上下文中保留 20 轮，会产生 100,000 个输入词元的处理成本。等到实际需要时再重新读取该文件一次，只需 5,000 个词元。驱逐它可节省 95,000 个词元。保留的代价高昂；缺页的代价低廉。

这种反转改变了优化目标。正确的表述不是“尽量减少缺页”，而是：

$$\min \sum_{p \in P} \left[C_{\text{keep}}(p) + C_{\text{fault}}(p) \right]$$

其中 $C_{\text{keep}}(p) = |p| \cdot T_{\text{resident}}(p) \cdot c_{\text{token}}$ 是将页 p 保留在内存中 T_{resident} 轮所产生的累计成本，而 $C_{\text{fault}}(p) = |p| \cdot c_{\text{token}}$ 是恢复该页的一次性成本（重新处理一轮）。盈亏平衡条件为：

$$|p| \cdot T_{\text{until\_next\_ref}}(p) \cdot c_{\text{token}} > |p| \cdot c_{\text{token}}$$

也就是说：如果某个页在超过一轮的时间内都不会被引用，就应当将其驱逐。这就是为什么 FIFO 在我们的系统中表现如此出色，尽管它在经典虚拟内存中是表现最差的替换策略：当保留成本高而缺页成本低时，默认采用积极驱逐就是正确的做法。只有为了避免驱逐下一轮马上就会用到的页，才需要精细设计替换策略。

倒置成本下的 ***Belady*** 最优置换算法。 Belady 最优算法 [1] 会驱逐下次引用时间最晚的页，从而使缺页总数最少。在倒置成本模型下，最优策略则有所不同：它必须将保留成本与缺页成本之和降至最低。对于一个下次引用时间遥远的页，应立即驱逐（以一次缺页为代价，节省多轮保留成本）。每轮都会被引用的页则绝不应驱逐（缺页成本等于一轮的保留成本，因此驱逐毫无节省）。倒置成本下的最优离线策略并非 MIN，而是一种成本加权变体：当页的预计保留成本超过其缺页成本时，就将其驱逐，无论是否会发生缺页。

这带来了一个实际影响：页的大小会影响驱逐优先级，而在经典虚拟内存中并非如此（因为所有页的大小都相同）。一个包含 10,000 个词元的文件，每轮的成本是一个包含 1,000 个词元的文件的十倍。除非访问频率足以证明其保留成本合理，否则应尽早驱逐较大的页。

非线性缺页成本。上述线性缺页成本 $C_{\text{fault}}(p) = |p| \cdot c_{\text{token}}$ 是一种简化。Transformer推理主要受自注意力支配，其复杂度随序列长度 $O(n^2)$变化。一次缺页需要额外进行一整遍推理——模型会发出工具_使用召回句柄的请求，恢复的内容则作为工具_结果消息返回，并触发对整个上下文进行第二遍推理。因此，实际缺页成本与当前上下文大小 n的 n^2 成正比，而不只是与页大小 $|p|$ 成正比。

这会产生一种违反直觉的策略梯度。在填充率较低时（例如 $n = 40$K），缺页成本较低，积极驱逐是正确的——额外一遍推理的二次成本并不高，而且缩小工作集可以降低后续所有轮次的 n^2 基础。填充率较高时（例如 $n > 100$K），缺页成本会变得高昂：额外一遍推理的计算量大约相当于$(n + |p|)^2 \approx n^2$ 词元。因此，随着上下文压力上升，驱逐策略应当更加保守——这与压力下积极驱逐的直觉恰恰相反。填充率较高时，系统只应驱逐那些它非常确信不会再被引用的内容。

基于对象的成本方差。与经典虚拟内存不同，经典虚拟内存中的所有页都是固定大小的块（通常为4KB），而Pichay管理的是可变大小对象：召回的张量可能是3行的摘要，也可能是200行的文件。这意味着工作集中不同对象的驱逐成本和缺页成本可能相差数个数量级。替换策略不能将所有驱逐候选一视同仁；它必须根据对象大小，同时权衡每轮保留成本和潜在缺页成本。结合二次缺页惩罚，这说明需要一种感知大小且对填充率敏感的替换策略，而经典虚拟内存文献中没有直接对应的策略。

缓存失效成本。 截至目前，成本模型仅考虑词元级成本（保留和缺页）。结构变更——消息数组中移除或替换块的折叠操作——还会产生额外成本：提示缓存失效。推理服务提供商会缓存重复请求的分词前缀

；当前缀发生变化时，缓存未命中，整个上下文都必须重新处理。

在生产环境中，将 12 轮引导对话压缩为一句摘要的折叠操作，导致缓存命中率在一轮内从 100% 降至 25%，随后随着新前缀稳定下来，下一轮又恢复到 100%。代价是对 ~105 K 个词元进行一次完整的重新计算——相当于数次缺页的成本。折叠操作虽然节省了 10KB 上下文，却使一个包含 100K 个词元的已缓存前缀失效；除非节省的空间能在后续足够多轮中持续发挥作用，以摊销这一次性重新计算，否则得不偿失。

这说明应当批量处理结构变更：先累积折叠候选，再一次性执行，从而只支付一次缓存失效成本，而不是每次操作都支付。也说明应避免频繁进行小规模折叠，转而采用低频、大规模折叠。

固定衰减。倒置成本模型也不支持永久固定。在当前设计（第 3.5 节）中，一次缺页会使某个页永久固定。但缺页只能说明该内容当时有用，并不意味着它永远有用。根据成本模型，固定强度应当衰减：自上次访问后每经过 K 轮，固定强度减半；当剩余固定生命周期的预计保留成本超过缺页成本时，该页便可被驱逐。这样既能实现类似最近最少使用的行为，并进行成本加权衰减，也能避免永久固定导致工作集在长会话中不断增长。

6.3 可推广性

我们测得的浪费模式是智能体架构带来的结构性后果：

- 任何在每次请求中都发送工具定义的系统，都会发送未使用工具的模式。具体比例取决于工具数量和使用分布，但这一机制具有普遍性。
- 任何将指令注入消息的系统都有重复的风险。Claude Code 的三重重复现象尤为严重，但并非独有。
- 任何在整个会话生命周期内都将工具结果保留在上下文中的系统，其放大效应都会与会话长度成正比。

我们的实证证据来自一个系统（Claude Code）和一位用户。结构性论证很有力——架构本身注定会造成浪费——但对其他工具进行比较测量，将有助于增强这一可推广性主张。

6.4 范围、非目标与效度威胁

范围。 本文评估智能体编码工作负载中的上下文浪费和按需分页行为，这类工作负载会持续保留工具输出和对话记录历史。本文的目标主张针对机制层面：当请求反复重新发送大型静态前缀和过时的工具输出时，驱逐与基于缺页的恢复能够减少输入处理的累计开销。

非目标。 我们不主张在所有助手、所有模型系列或所有任务领域中都能普遍降低某个百分比。我们不评估前沿推理基准测试、自主规划质量，也不在产品规模上评估人类偏好。我们也不主张 L1 扩展（更大的上下文窗口）已经过时；我们的主张是，若缺少分层管理，L1 扩展并不完整。

效度威胁。外部效度受限于单用户语料库以及单一主要助手实现。内部效度风险包括提示词漂移、跨版本的工具集变更，以及会改变热集组成的工作负载阶段效应。为缓解漂移，我们针对主要数字报告一个冻结队列，并提供可执行的重新计数工具以及带时间戳的实时快照产物，以确保可审计性。

6.5 质量：LLM评判的等价性检查

该干预不得降低输出质量。因此，我们进行配对式等价性检查：若结果大体相当而计算量下降，则该干预是有用的。

协议。我们从语料库中选取 18 个会话（426–1,354 条消息，340K–718K 个上下文字符）。对于每个会话，我们在对话的 65–75% 处构建配对上下文：基线 保留所有消息；处理组将 20 条消息近期窗口之外已消耗的工具结果替换为墓碑标记，以模拟 Pichay 的驱逐行为。两种条件接收相同的续写提示（下一条用户消息）。两种条件下的输出均由 Sonnet 4 生成。

由三位大语言模型评审组成的评审组（一位 Sonnet 4、两位 Haiku 4.5）会从正确性、完整性和连贯性（1–5 分制）方面评估每组配对输出。评审还会给出盲测偏好（A、B 或平局），并判断自己能否识别出哪个输出的上下文有所缩减。每位评审看到的 A/B 分配均随机确定，以避免位置偏差；汇总前，评分会重新映射为基础组/处理组。

表 9. 非劣效评估：18 个会话，54 个评判结果。处理组会从 20 条消息的近期窗口之外移除已消耗的工具结果（平均压缩率为 48%）。

指标	基线	处理组
评审偏好	15（28%）	20（37%）
Ties	19（35%）	
平均正确性	3.89	3.74
平均完整性	3.59	3.59
平均连贯性	3.74	3.69
最高得分 Δ	0.15	
检测率	57% (p = 0.14, n.s.)	

结果。 评判者更常偏好处理组 而非基线（37% 对 28%），另有 35% 的平局。完整性得分

相同；在 5 分制下，正确性和连贯性最多相差 0.15 分。检测率（57%）并未显著高于随机水平（二项式 $z = 1.09$, $p = 0.14$, $n = 54$）。

失败模式。 两个会话（11%）产生了退化的处理组输出（零字符）：续写提示隐式引用了仅存在于已标记删除的工具结果中的内容。这些是真正的驱逐伤亡。失败模式很具体：用户的提示按内容而非按名称引用了最近已消费的工具结果。引用感知驱逐策略会检测到此类前向依赖；当前启发式（已消费 +近期窗口之外 = 可驱逐）则不会。

注意力集中效应。 在若干会话中，三位评判者都更偏好处理组输出。移除一半上下文并非仅仅保持质量——反而提升了质量。这与注意力稀释假说一致：Transformer 注意力将权重分配给上下文中的所有词元；无关词元（过时的工具结果、已消费的中间输出）会稀释对重要词元的注意力。驱逐它们可将注意力集中于信号。该效应在工具结果密度较高的会话中最强，其中被驱逐的内容确实是噪声。我们将此记为初步证据；要严格确立该效应，需要使用更大的 N 进行受控注意力分析。

6.6 计算与能耗影响

我们的语料库中有 9.7 亿词元属于可处理浪费，这会带来三个计算成本维度：

输入处理。 缓存命中率为 93.5% 时，大多数节省下来的词元都对应缓存读取（9.07 亿词元，成本较低）。然而，6300 万词元属于缓存创建或未缓存内容——它们需要完整的前向传播计算，而这些计算本可避免。

注意力开销。 对于每个输出词元，模型都要针对完整上下文计算注意力。将平均上下文中的 17,913 个词元移除（平均上下文长度为 82,061 个词元），可使每个输出词元的注意力计算量减少 21.8%。在整个语料库中，这相当于消除 850 亿个令牌间注意力对。

吞吐量。 上下文中的每个词元都需要占用 KV 缓存内存。对于大型模型，每个词元的 KV 缓存占用量约为数 MB。每个请求的上下文越短，所需的 KV 缓存内存就越少，因此每块 GPU 可以同时处理更多请求。粗略而言，上下文减少 21.8%，意味着在同一 GPU 集群上可同时服务的用户增加 21.8%。

这正是改变基础设施经济效益的关键论点。节省的不只是“每次请求的成本更低”，而是“每台机器能够处理更多请求”——这决定了是否需要新建数据中心。

跨会话累积。 这些每次调用的节省会不断累积。在第 N 轮避免的浪费并不只是节省了一次

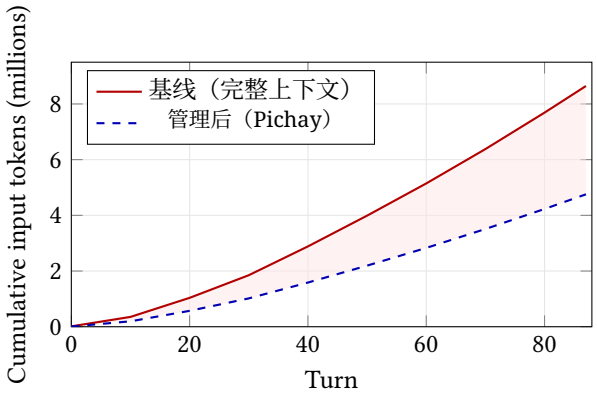

图2. 一场持续 88 轮的会话中累计处理的输入词元。没有上下文管理时，每一轮都会重新处理此前的所有内容，导致累计成本呈超线性增长。Pichay 的驱逐使累计词元开销降低了 45%（按 Opus 定价为 11.67 美元）。阴影区域表示浪费的计算资源。

；它还会从后续每一轮的上下文中消失，否则这些内容会在 $O(n^2)$ 的注意力成本下被重新处理。因此，在一次会话中，总计算节省相对于浪费比例呈超线性增长，并且会随会话长度增加。对于一个包含 100 轮、具有 21.8% 可处理浪费的会话，累计注意力节省并不是整个会话计算量的 21.8%，而是显著更高，因为携带不必要词元所产生的二次成本会在每次后续调用中不断累积。对基础设施的影响由此直接显现：最快的词元，就是那些你从未处理过的词元。

图2展示了 Pichay 在一场持续 88 轮的实时会话中产生的累积效应。累计基准成本（每轮都处理全部上下文）达到 860 万个输入词元；管理后成本为 480 万，累计减少了 45%——这远高于逐轮压缩率，因为每个被驱逐的词元在此后的每一轮中都不再出现。

舰队规模外推。我们的语料库涵盖一位用户四个月的使用情况。要进行全舰队范围的预测，需要掌握每位用户的词元消耗数据，这些数据由推理服务提供商持有，但尚未公开。外推框架很简单：N 用户 $\times$ 970M 词元/用户 =可寻址总量浪费。其机制（工具模式浪费、内容重复、结果放大）由架构决定，因此，即使词元总量有所变化，每位用户的浪费比例也应相对稳定。

7 未来工作

***L3*评估。** 崩溃机制（第 3.9 节）已实现，但尚未进行大规模评估。待解决的问题包括：与系统发起的驱逐相比，由模型编写的压缩能恢复多少上下文？声明的

折叠摘要中的损失能否支持准确的缺页决策（模型能否仅凭摘要判断自己是否需要原始内容）？将 20 轮对话脚手架替换为单个结果句会对质量产生什么影响？这些都需要在不同长度和任务复杂度的会话中进行纵向测量。

成本感知驱逐压力。当前的驱逐策略使用基于空间的阈值（上下文窗口容量）。倒置成本模型主张采用基于成本的压力：23%的内存压力听起来不高，但按推理定价计算，每轮45,000个词元却是实实在在的成本。驱逐压力应以token-轮次（页大小与预计剩余生命周期的乘积）来计量，而不是以分数容量来计量。这将驱逐的考量从“我们的空间是否快用完了？”重新表述为“这一页是否配得上其保留成本？”

成本加权钉住衰减。钉住页不应永远驻留。一次缺页只说明该内容在当时被需要，而非永远需要。我们提出成本加权衰减：自上次访问起，钉住强度每 K 轮减半。当某一页的预计保留成本超过其缺页成本时，该页即可被驱逐。在低内存压力下，这一机制较为宽松（钉住会持续）。在高压力下，冷钉住会首先被释放。这带来带有成本加权的类似最近最少使用的行为，并防止永久钉住在长会话中导致工作集单调增长。

阶段感知驱逐。规划与执行具有不同的工作集特征。规划需要广泛的同时上下文（一次持有许多文件）；执行则是顺序的（读取、编辑、继续前进）。代理可以根据请求模式检测阶段——大量 Read 且没有 Edit 表明处于规划阶段——并据此调整驱逐阈值。

每连接隔离。当前实现为所有连接使用单一 PageStore。子代理会话与主会话共享驱逐状态，从而导致交叉污染。每连接隔离将为每个会话提供各自的驱逐历史、缺页跟踪和钉住状态。

轨迹驱动仿真。我们从会话记录中提取引用串（页访问序列），并在不同替换策略下重放它们。这使得能够针对已记录的工作负载评估策略，而无需进行按实时推理计价的实验。倒置成本模型（第 6.2 节）意味着最优离线策略并非 Belady 的 MIN，而是一种最小化总保留成本与缺页成本之和的成本加权变体。我们计划推导并评估这一成本最优离线下界，并与 MIN 进行比较。

跨会话访问模式预测。盈亏平衡公式需要估算 $T_{\text{until\_next\_ref}}$——即一页再次被引用之前相隔的轮数。在单个会话中，这只能根据局部模式进行估算。跨会话时，预测就变得可行：编程会话会呈现规律的访问模式（了解项目结构的环境了解读取、对工作文件进行迭代编辑、定期参考配置）。基于跨会话引用字符串训练的马尔可夫模型，可以预测每页每轮的再引用概率，从而支持成本最优的驱逐决策。数据收集几乎不费成本：代理已经记录了每次驱逐、每次缺页和每次工具调用。所需的是一个持久、可查询且能跨越会话边界保存访问模式的存储——这正是跨会话记忆系统的作用。

协作式扩展。幻影工具在模型与代理之间打开了一条旁路信道。当前的词汇——释放和缺页——十分精简。可扩展的内容包括：预取提示（模型发出即将需要某个工作集的信号）、驱逐类别（将一组页作为一个整体驱逐）、可变保真度墓碑（使用 AST 骨架，而非单纯的路径标记），以及临时标志（工具结果在提取后标记为立即驱逐）。这些扩展都增强了模型向内存管理器传达认知状态的能力。

超越分页：对象寻址内存。当前系统管理的是按文件路径寻址、粒度固定的页。但大语言模型进行内存管理时，自然的单位是语义对象——一个对话阶段、一项设计决策或一次调试会话。这些对象的大小各不相同，彼此之间存在关联，而且可以压缩为多个保真度级别，而非像硬件页那样只有驻留/驱逐两种状态。驱逐墓碑会变成压缩摘要，兼具页表项（指向后备存储的检索句柄）和缓存行（包含足够的语义内容，无需触发缺页即可回答查询）的功能。摘要中声明的损失会告知模型，仅凭摘要无法回答哪些问题，从而帮助模型决定是否触发缺页。后备存储支持查询——模型可以针对已驱逐的内容提问，而不必将其完整载入内存。这使抽象从按块寻址的分页转向对象寻址内存管理：驱逐由协作完成，压缩由模型编写，后备存储则是数据库，而不是交换分区。

比较测量。对其他智能体工具（Cursor、Copilot、Windsurf、Continue.dev）进行仪器化，以测量相同的指标。结构性论证预测会出现相似的浪费模式；实证确认将增强可推广性。

8 结论

大语言模型的上下文窗口并非内存，而是 L1 缓存——在本应受管理的层级结构中，它只是一个容量小、速度快、成本高的层级。业界却将其视为整个内存系统，没有 L2、没有分页、没有驱逐策略，也没有工作集估算。结果是，在测量的 44.5 亿个词元中，结构性浪费高达 21.8%；模型还会以 84.4× 倍的放大效应，重新处理过时内容。

Pichay 证明，为 20 世纪 60 年代物理内存开发的虚拟内存抽象，同样适用于大语言模型上下文窗口。采用简单先进先出策略的按需分页，在稳态使用中可恢复 36 个百分点的上下文容量。故障驱动固定可减少稳态使用时工作集内容的重复缺页。检索句柄原本是为节省空间而设计的标记，却能充当延迟绑定的锚点，模型无需指令便能理解。渐进式压力区和协作清理标签赋予模型自行管理内存的能力——这种能力在硬件内存层次结构中没有对应物。

该系统无需修改模型、客户端或推理 API。它以透明代理的形式实现，并已部署于生产环境。你正在阅读的这篇论文就是通过它撰写的。

更深层的贡献在于这一架构性观察：大语言模型系统需要的不是一个庞大的上下文窗口，而是一套受管理的层次结构——从生成窗口（L1）一直延伸到持久的跨会话内存（L4）——各层由驱逐策略和缺页机制连接起来。前三个层级已部署；前两个已完成评估。L3——模型发起的对话压缩——是最新的机制，其评估仍在进行中。剩余层级的解决方案已见于操作系统文献。接下来要做的，就是将它们连接起来。

作者身份与人工智能贡献

本文由一位人类系统研究员与一个人工智能系统（Anthropic 的 Claude Opus）共同撰写。人类作者贡献了内存层次结构见解、研究方向、实验设计和编辑判断，其中包括指出上下文管理就是内存管理，这一观察重新界定了整个项目。人工智能贡献了代码（分析工具、代理实现、分页系统）、数据分析和论文初稿。

Pichay 由正在研究的系统构建——Claude Code 通过自身的代理运行——这篇论文也是通过它撰写的。第 5.7 节中的评估数据是该系统自身开发过程中的副产品。我们认为这种自指性是一项优势：系统能够在数百轮交互中测量、优化并维持自身上下文使用情况，这是干预措施切实可行的最有力证据。

所有工具和数据均为开源，可访问：https://github.com/fsgeek/pichay。本文引用的论文快照是提交 b56701a 上的发布版本 v0.1.0-paper。不可变的存档快照可通过 Zenodo 数字对象标识符获取：https://doi.org/10.5281/zenodo.18930122。

数据快照与计数协议

为了在实时语料库不断演变时确保计数可复现，我们将论文中的数字（本草稿使用的冻结队列）与实时重新计数的结果分开。计数协议在代码仓库中进行版本管理：docs/corpus_protocol.md。

计数工具。会话计数由 tools/corpus_count 执行它将 Claude Code JSONL 文件分类为主会话、子代理、压缩、提示_建议和其他，并报告原始总数以及内容哈希去重总数。

重新计数命令。

./工具/reproduce_paper_counts.sh
这会将带时间戳的快照 JSON 写入 paper/data/corpus_snapsh

五分钟复现流程。从代码仓库根目录开始：

./工具/reproduce_paper_counts.sh python3 工具/check_paper_numbers.py \ --paper paper/main.tex \ --snapshot paper/data/语料库_snapshot_20260307_live.json latexmk -pdf paper/main.tex
预期输出：（1）paper/data/ 下的一份新快照 JSON，（2）一份论文数字一致性报告，以及（3）paper/main.pdf。在典型笔记本电脑上，此路径可在数分钟内完成；重新计数步骤受 I/O 限制，取决于本地语料库大小。

当前实时重新计数产物（***2026-03-07***）。paper/data/语料库_sna
~ ~根目录：/.claude/projects 和 /projects/yanantin/tmp/ubuntu-大小筛选条件 min_size=10000，以及来自 /projects/yanantin/tmp/claude-desktop/conversations.jso 的 Claude Desktop 摘要~

在本论文版本中，主要结果采用冻结的队列，以避免不同修订版本之间出现漂移；同时提供实时重新统计的产物，以确保可审计性。

参考文献

[1] Laszlo A. Belady. 1966. 面向虚拟存储计算机的置换算法研究。*IBM Systems Journal* 5, 2 (1966), 78–101. [2] deepset团队. 2026. 上下文工程：提示工程之后的下一个前沿。
h
t
t
p
s://
w
w
w.
dee
pset.ai/blog/context-engineering-the-next-frontier-beyond-prompt-engineering. 访问于2026年3月。[3] Peter J. Denning. 1968. 程序行为的工作集模型。*Commun. ACM* 11, 5 (1968), 323–333. [4] Peter J. Denning. 1970. 虚拟内存。*ACM Computing Surveys (CSUR)* 2, 3 (1970), 153–189. [5] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, 和 Lili Qiu. 2024. LongLLMLingua：通过提示压缩加速并增强长上下文场景中的大语言模型。收录于计算语言学协会第*62*届年会论文集（第*1*卷：长篇论文）。1658–1677. [6] Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, 和 Saravan Rajmohan. 2025. ACON：优化长程大语言模型智能体的上下文压缩。arXiv:
2510 00615 [ ] 2510 00615 . cs.AI
[7]
https://arxiv.org/abs/ 。Sanjay Kariyappa 和 G. Edward Suh. . SideQuest：面向长程智能体推理的模型驱动 KV缓存管理。arXiv:
2602 22603 [ ] 2602 22603 . cs.AI https://arxiv.org/abs/ 。
[8]
[
T Kilbum, D. B. G. Edwards, M. J. Lanigan, 和 F. H. Sumner. n. d.]. 一级存储系统。收录于*IRE Transactions, EC-Denning, Peter J.*"论程序行为建模"，弗吉尼亚州阿灵顿：*AFIPS Press*：春季联合计算机会议论文集，第40卷。937–944.[9] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, 和 Yaroslav Zharov. 2025. 复杂性陷阱：对于智能体上下文，简单的观察掩码与大语言模型摘要同样高效

管理。arXiv:2508.21433 [cs.SE] https://arxiv.org/abs/2508. 21433[10] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, 和 Shenghua Liu. 2025. 大语言模型上下文工程综述。arXiv: 2507 13334 [ ] 2507 13334 . cs.CL [11] https://arxiv.org/abs/ 。Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, 和 Joseph E. Gonzalez. 2024. MemGPT：迈向作为操作系统的大语言模型。arXiv:2310.08560 [cs.AI] https://arxiv. org/abs/2310.08560[12] Prithvi Rajasekaran, Ethan Dixon, Carly Ryan, 和 Jeremy Hadfield. 2025. 面向人工智能智能体的有效上下文工程。 https:// www.anthropic.com/engineering/effective-context-engineering-for-ai-agents. 访问于2026年3月。

[13] Cosmo Santoni。2026年。上下文记忆虚拟化：面向大语言模型智能体的基于有向无环图的状态管理与结构无损修剪。arXiv: 2602 22402 [ ] 2602 22402 。cs.SE [14] 2026 https://arxiv.org/abs/ 。Nikhil Verma。。主动上下文压缩：大语言模型智能体的自主记忆管理。arXiv:2601.07190 [cs.AI] https: //arxiv.org/abs/2601.07190 [15] Yuhang Wang、Yuling Shi、Mo Yang、Rongrui Zhang、Shilin He、Heng Lian、Yuting Chen、Siyu Ye、Kai Cai 和 Xiaodong Gu。2026年。SWE-Pruner：面向编程智能体的自适应上下文剪枝。arXiv: 2601 16746 [ ] 2601 16746 。cs.SE [16] 2025 https://arxiv.org/abs/ 。Simon Willison。。上下文工程。 https://simonwillison.net/ 2025/jun/27/context-engineering/。访问于2026年3月。[17] Philip Zeyliger。2026年。代价高昂的二次增长：大语言模型智能体成本曲线。https://blog.exe.dev/expensively-quadratic。访问于2026年3月。