arXiv:2603.28795v1 [cs.OS] 24 Mar 2026

StepCache：面向大语言模型服务的步骤级复用、轻量级验证与选择性修补

Azam Nouri

林肯大学科学、技术与数学系，美国

摘要

我们针对这样的大语言模型服务工作负载开展研究：重复请求具有共同的解题结构，但局部约束条件有所不同（例如输出模式、变量名称或数值常量）。以往的缓存方法通常复用完整响应（语义缓存）或模型内部的 KV/前缀状态；前者在发生局部变化时往往不够稳健，后者则与特定后端紧密耦合。我们提出 **StepCache**，一种与后端无关的步骤级复用层，可将输出拆分为有序步骤，检索匹配度最高的缓存请求，使用轻量级任务感知检查验证步骤，并通过选择性修补仅重新生成失败区域。StepCache 还支持针对 JSON 的严格结构化输出约束（单步提取、必需键约束、单次修复），以及针对语义变化的保守跳过复用回退机制。对于线性方程，StepCache 通过有界修复循环将验证转化为纠正，并提供确定性回退机制，确保后端模型失败时仍能保证正确性。

在一个仅使用 CPU、针对数学和 JSON 变体的高扰动微基准测试中（每个任务有 10 个基础提示，每次扰动有 k=3 个变体；每个随机种子对应 222 个评估请求），对三个随机种子（ 42–44 ）的结果取平均后，StepCache 将平均延迟从 2.13 秒降至 0.67 秒（中位数：从 2.42 秒降至 0.01 秒；第95百分位数：从 3.38 秒降至 3.30 秒），将令牌总用量从 36.1k 降至 27.3k （每个请求 162.7→123 个令牌），并在任务专用检查和拼接输出完整性检查下，将端到端正确性从 72.5% 提升至 100%。在所有请求中，79.7% 走仅复用快速路径，5.4% 需要修补，14.9% 触发跳过复用。

1 引言

大语言模型（LLM）正越来越多地部署在交互式系统中，在这类系统里，端到端延迟和服务成本是首要约束条件（例如代码助手、智能体工作流和结构化数据生成）。近期的大语言模型服务研究主要聚焦于后端效率，尤其是 GPU 内存管理以及 KV 缓存的调度；同时也关注缓存策略，以避免为重复或高度相似的请求重新计算输出。

尽管取得了这些进展，现有缓存机制通常以粗粒度运作，表现为要么全部复用，要么完全不复用：

- **语义响应缓存**会在新提示在嵌入空间中足够相似时，返回先前生成的完整响应，有时还会辅以验证或置信度阈值。
- **前缀/KV 缓存**会复用模型内部状态以处理重复前缀，但它依赖于特定模型、分词器和上下文，主要用于加速重复提示前缀，而非复用答案的部分内容。

在许多实际工作负载中，请求只是部分相似：解决方案的结构保持稳定，但由于出现了新的约束或参数，需要进行局部的小幅修改。例如，向 JSON 模式中添加必需键（“添加键 d”）、更改目标变量（求解 y 而非 x），或修改熟悉模板中的常量（例如，更改 $2x + 3 = c$）。在这种情况下，复用整个响应可能导致错误，而完全重新生成则会重新计算原本可以复用的解决方案中的大量内容。这一差距促使我们采用更细粒度的复用单元，以便在局部修改时仍能确保正确性。

我们提出 **StepCache**，这是一种与后端无关的复用层，它将答案视为由步骤组成的有序序列，并支持选择性重新生成。StepCache 将先前的输出分段为步骤，检索最匹配的缓存请求，通过轻量级任务感知检查验证缓存步骤，并通过连续块修补仅重新生成失败区域。由于 StepCache 在模型运行时之上运行，且不依赖模型内部的 KV 状态，因此可作为即插即用的优化层部署在现有服务引擎之前，适用于不同后端。

与此同时，在存在低成本验证器的场景中，StepCache 能提供最强的正确性保证（例如，解析包含必需键的结构化 JSON，或检查线性方程一致性）。对于开放式

生成，如果缺乏可检查的不变量，验证必然较弱或成本较高（例如使用大语言模型评审器），而 StepCache 可能会退回到较为保守的策略，例如跳过复用。为开放式文本设计广泛适用且低成本的验证器，是未来工作的重要方向。

贡献。

• **步骤级复用**。 我们为大语言模型服务引入一种步骤粒度的缓存抽象：检索一条最匹配的历史请求，保留步骤顺序，逐步验证，并仅对失败区域进行选择性重新生成，而非重新生成整个响应。

• **采用安全回退机制的选择性修补**。 我们针对依赖密集型推理开发了连续块修补，并制定了一种自适应跳过复用 策略：当不一致信号表明复用可能无法奏效时，回退到完全重新生成。

• **任务感知验证与修复**。 我们为两个有代表性的任务类别实现了轻量级验证器和有界修复循环：结构化 JSON 生成（单步提取、必需键约束、单次修复）以及线性方程求解（数学一致性检查、变量感知验证、单次修复）。

• **带检测功能的原型与评估**。 我们提供了端到端原型，详细记录各项指标（调用次数、令牌数、延迟），并设计了一个以扰动为主的微基准测试，用于量化复用率、修补率、鲁棒性和尾部行为。

2 背景与相关工作

大语言模型服务与 KV 缓存管理。 大量研究通过优化模型运行时，尤其是注意力 KV 缓存的内存管理和调度，提升大语言模型服务效率。PagedAttention（用于 vLLM）可减少内存碎片，并支持高吞吐量解码时对 KV 状态进行高效分页 [1]。近期系统进一步通过压缩和淘汰策略降低 KV 开销 [3]，或重新配置 KV 分配，以满足长上下文工作负载下的尾延迟／服务等级目标 [4]。StepCache 与这些方法互为补充：它作为应用层复用机制，在服务后端之上 运行（包括步骤检索、验证和选择性重新生成）；当重复请求共享部分结构时，它可与 KV 缓存优化结合，同时降低端到端延迟和后端计算开销。

工作负载特征与尾延迟。 实际服务工作负载具有突发性、偏斜以及重尾延迟分布。BurstGPT 提供的轨迹和分析强调了尾部指标以及负载下鲁棒性的重要性 [2]。我们的评估报告平均值、中位数和 P95 延迟，并明确区分快速路径（复用）与慢速路径（修补或跳过复用回退）。将 StepCache 与公开工作负载上的轨迹回放相结合，是自然的下一步。

语义响应缓存与验证。 语义缓存系统通过在新提示足够相似时返回先前生成的响应来加速应用，通常采用嵌入检索，并可选择进行验证。GPTCache 是一种具有代表性的开源语义缓存 [5], ，近期研究也提出了经验证的语义缓存机制，旨在复用缓存输出时提供正确性保证（例如，vCache [6] 和分层验证缓存 [7]）。这些方法通常将整个响应视为复用单元，因此采取非此即彼的决策：要么原样接受缓存响应，要么由系统重新生成新响应。StepCache 面向一种互补的应用场景：提示共享稳定的解答骨架，但局部约束有所不同。它将有序步骤作为复用单元，进行逐步轻量验证，并采用连续块修补，仅重新生成第一个失败步骤及其下游依赖项；当不一致信号表明复用可能无益时，StepCache 会保守地触发跳过复用，并回退到完全重新生成。

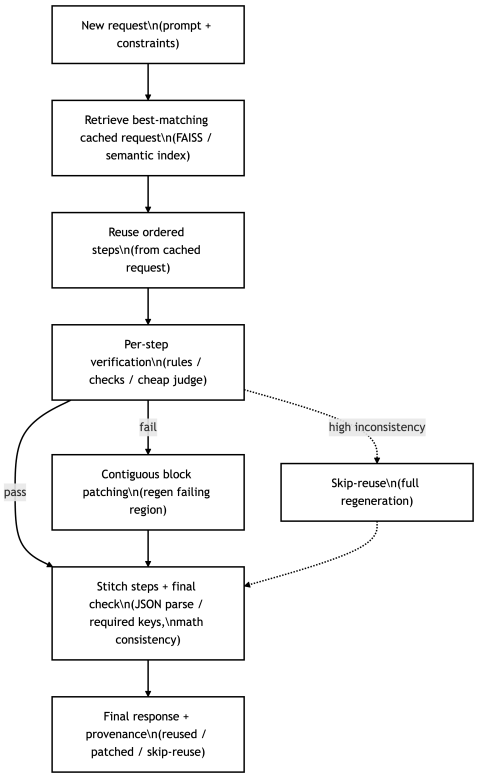

图 1：StepCache 流程：检索、逐步验证、通过连续块修补进行选择性重新生成，以及拼接输出完整性检查。

3 StepCache 设计

3.1 概述

图2简要展示了 StepCache 的概念，而图1展示了端到端流程。首次收到请求时（或在预热阶段），StepCache 会将提示转发给后端大语言模型，将生成的输出分段为有序步骤列表，并存储 (i) 用于检索的提示嵌入，以及 (ii) 带有轻量级元数据（任务类型和约束条件）的步骤序列。

对于新请求，StepCache 会计算提示嵌入，检索匹配度最高的单个缓存请求（提示间相似度），并在新提示和约束条件下评估缓存步骤序列。对于每个步骤，StepCache 会：

- **复用**通过验证的步骤；
- **修补**未通过验证的步骤：仅重新生成失败区域，通常将其作为一个连续块，以遵循步骤依赖关系；
- **回退到**完全重新生成（跳过复用），当不一致信号表明复用可能需要大量修补或降低正确性时。

随后，StepCache 将生成的步骤列表拼接成最终响应，并执行任务级完整性检查（例如，JSON 解析 + 必需键；数学解答一致性）。如果拼接后的输出未通过这项最终检查，StepCache 就会执行有次数限制的任务专用修复（在我们的原型中，针对 JSON 和数学任务各执行一次），并在返回答案前重新验证。

StepCache 流程：
(1) **嵌入** 提示 → (2) **检索**最佳缓存请求 → (3) **验证** 每个缓存步骤→ (4) **复用** 通过的步骤
+ **修补** 失败的步骤（阻止）或 **跳过复用** → (5) **拼接** → (6) **最终检查** （数学 /JSON ） +
有次数限制的**修复** → 答案 +逐步溯源信息。

图2：StepCache 概览。StepCache 复用已验证的步骤，并仅选择性地重新生成失败的步骤。

3.2 缓存单元：步骤

StepCache 将响应视为一个有序的步骤序列。默认情况下，分段采用启发式且与任务无关的方法：我们根据段落边界（双换行）、显式编号（例如“步骤 1”）和列表分隔符进行拆分。对于结构化输出任务，分段则会考虑任务。具体而言，对于 JSON生成，我们强制采用**单步分段**：StepCache 从模型输出中提取第一个语法有效的 JSON 对象/数组（若存在代码围栏和外围文本，则将其移除），并将该 JSON负载缓存为唯一的步骤。这种设计可确保下游验证和修补以单个结构化单元为对象进行。

对于每个缓存请求，StepCache 会存储：

- 一个用于提示词间检索的**提示嵌入**，
- 为该请求生成的**有序步骤文本**，
- **约束元数据**（例如，任务类型、必需键），
- 可选的**工具输出**（在我们的微基准测试中未使用），
- 用于核算的**溯源信息和时间**信号（复用/修补/跳过决策）。

3.3 检索

在 MVP配置下，StepCache 会根据单个缓存请求与提示嵌入的相似度，检索最匹配的请求，并按原始顺序复用其步骤列表。为单独考察步骤级复用的效果，我们不会混合来自不同缓存请求的步骤。我们使用 FAISS 实现近似最近邻检索，并将每个请求的元数据存储在本地数据库 [10] 中。

3.4 验证

StepCache 使用轻量级的任务感知验证，判断缓存步骤在新的提示和约束条件下是否仍然有效。我们的原型为两个具有代表性的任务族实现了**基于规则的验证器**。

数学（线性方程）。对于形式为 $a \cdot v + b = c$（其中 v 是目标变量）的提示，StepCache 会解析 (a, b, c, v)，计算预期解 $v^\star = (c - b)/a$，并标记与这些值相矛盾的缓存步骤。验证器会检查以下内容：

- 错误的最终赋值（例如，$v = 5$当 $v^\star = 6$ 时），
- 错误的中间等式（例如，$av = N$ 其中 $N \neq c - b$），
- 错误的方程常数陈述（例如，$av + b = N$ 其中 $N \neq c$）。

如果第一个不一致出现在步骤索引 i 处，StepCache 会保守地将步骤后缀 $i..$end 标记为失败，从而支持遵循步骤依赖关系的连续块修补。

JSON（必需键）。StepCache 通过解析缓存的 JSON，并可选择强制执行必需键约束条件，来验证其有效性。如果解析失败或缺少任何必需键，该步骤的验证就会失败。此验证器与严格修补提示和一次有界修复尝试相结合，用于结构化输出恢复。

3.5 选择性修补与修复

算法 1 概述了 StepCache 的推理过程。给定检索到的步骤序列，StepCache 首先通过任务感知验证（适用时参见第 3.4 节）识别失败的步骤。StepCache 不会重新生成整个响应，而是执行选择性重新生成：复用已验证的步骤，仅重新生成在新提示和约束条件下恢复正确性所需的最小区域。

失败 发生时，StepCache 会根据任务结构采用两种修补策略之一。 re:

• **连续块修补（数学）**。对于方程求解等依赖密集型推理，StepCache 会从第一个不一致的步骤索引 i 开始，修补直至末尾的步骤后缀。每次修补调用都会包含一条数学状态提示，其中包括（$a,\ b,\ c,\ v,\ v^{\star},\ c-b$）（变量名、预期解和预期中间值），以避免复用过时的常量，并确保重新生成的步骤在数值上保持一致。

• **严格结构化修补（JSON）**。对于 JSON 任务，StepCache 会使用专用提示来修补（单个）结构化步骤，要求输出仅包含有效 *JSON*（不含 Markdown 或解释），强制满足任何必需键约束，并提供模式示例。修补后，StepCache 会重新解析输出并检查必需键；如果验证失败，则最多再尝试一次修复，并使用错误反馈（解析错误或缺少键）。

保守跳过复用与确定性回退机制（数学）。当语义变化导致大部分缓存解答失效时（例如，更改方程中的常数），选择性修补并非总是划算。因此，StepCache 对数学任务采用保守的跳过复用策略：(i) 如果新提示与检索到的缓存请求中的方程解析状态（$a,\ b,\ c,\ v$）不同，StepCache 将绕过复用；(ii) 如果第一个不一致的步骤是步骤 1（即没有缓存步骤通过验证），或者不一致步骤的比例超过阈值（原型中设为 0.5），StepCache 也将绕过复用。在基准测试中，我们还为值变更案例标记了强制跳过复用约束，以隔离语义变化的影响。

为确保此类可检查任务的正确性，StepCache 会尝试进行有限次修复；如果检查仍未通过，则返回一个基于解析出的方程计算得出的最简确定性解答，形式为“$v = v^{\star}$”。这样一来，低成本验证便成为一种能够保证正确性的线性方程回退机制。

4 实现

我们将 StepCache 实现为一个精简的 Python 层，置于兼容 OpenAI 的聊天补全 API 前端。该设计有意做到与后端无关：StepCache 不会访问模型内部的 KV 状态，只需要标准的请求/响应 I/O，以及在可用时提供的令牌使用元数据。因此，StepCache 可以部署在 GPU 推理服务引擎（例如 vLLM）或本地/远程模型端点前端，而无需修改模型运行时。

对于数学任务，StepCache 会通过稳健的提示解析来检测 (a、 b、 c、 v) 中的语义变化，并保守地触发跳过复用；如果经过有限修复尝试后任务检查仍然失败，它会根据解析出的方程返回一个最简、确定性的解“$v = v^{\star}$”。

组件。

• **嵌入**。 我们使用 SentenceTransformers all-MiniLM-L6-v2 计算提示嵌入，该模型基于 Sentence-BERT 风格的双编码器和 MiniLM 蒸馏 [8, 9]。

• **索引与存储**。 我们使用 FAISS 进行近似最近邻检索，并将每个请求的元数据（步骤列表、任务约束条件、计数器）存储在本地数据库中 [10]。

• **后端模型**。 在微基准测试中，我们使用通过兼容 OpenAI 的端点提供服务的本地小型模型（我们的实验中使用 Qwen2.5-3B） [11]。将后端切换为 GPU 引擎无需更改 StepCache 语义。

• **监测与统计**。 StepCache 对每次后端调用进行结构化日志记录，并维护基线生成、缓存命中复用、验证、补丁调用和跳过复用回退的明确计数器。基准测试记录每个请求的延迟和令牌用量，并报告平均值和尾部统计数据（中位数、第95百分位数），从而清楚地区分快速路径复用与慢速路径修补/回退行为。

算法 1 StepCache 推理（简化版；步骤从 1 开始编号） 要求：提示 p、任务类型 t、约束条件 C、缓存 $\mathcal{D}$ 1: $e \leftarrow$ 嵌入(p) 2: (p', S) $\leftarrow$ 最佳检索($\mathcal{D}, e$) $\triangleright$ $S = [s_1, \ldots, s_m]$ 3: 如果 S 为空，则 $\triangleright$ 缓存未命中 4: $A \leftarrow$ 大语言模型生成(p)； $S \leftarrow$ 分段(A)；存入 $\mathcal{D}$；返回 A 5: 结束如果 6: 如果 $t =$ 是数学，则 7: 如果 C 表示强制跳过，或 MathState(p) 与 $\neq$ MathState(p') 不一致，则 8: 返回大语言模型生成(p) $\triangleright$ 语义发生变化时保守跳过复用 9: 结束如果 10: $i \leftarrow$ 首个不一致索引(S, p) $\triangleright$ $i \in \{1, \ldots, m\}$ 或 $\perp$ 11: 如果 $i \neq \perp$ 且（$i = 1$ 或不一致比例(S, p) ≥ 0.5），则 12: 返回大语言模型生成(p) $\triangleright$ 跳过复用 13: 否则，如果 $i \neq \perp$，则 14: 将步骤 $i..m$ 标记为失败 15: 结束如果 16: 结束如果 17: 对 $j = 1$ 到 m 执行 18: 如果 s_j 未标记为失败，则 19: $v_j \leftarrow$ 验证(s_j, p, t, C) 20: 如果 $v_j =$ 失败，则将 j 标记为失败 21: 结束如果 22: 结束如果 23: 结束循环 24: $S' \leftarrow$ 复用通过的步骤；修补失败的步骤（分块/结构化） 25: $A' \leftarrow$ 拼接(S') 26: 如果最终检查(A', p, t, C) $=$ 失败，则 27: 至多执行一次任务专用修复并重新检查 28: 如果仍然失败且 $t =$ 是数学，则 29: 返回确定性求解(p) 30: 结束如果 31: 结束如果 32: 返回 A'

5 评估

5.1 实验设置

我们使用仅使用 CPU 的微基准测试评估 StepCache，该测试旨在考验扰动鲁棒性：每个基础任务都会生成多个变体，这些变体在保持整体意图的同时修改局部约束条件。我们纳入了两个有代表性的任务族:

- **数学（线性方程）**。在改述以及会改变等式右侧常数的语义扰动（值变更）下，求解形式为 $a \cdot v + b = c$（例如 $2x + 3 = 13$）的提示。
- **JSON（结构化输出）**。在改述以及会添加必需键的约束扰动（键变化）下，生成包含指定键的 JSON 对象。

扰动级别包括低/中/高程度的改述（以及约束条件添加），还包括值变更和键变化。每次运行都包含一个用于为缓存播种的预热阶段，以及一个用于测量检索、验证、修补和回退行为的评估阶段。

我们比较:

- **基线：** 对每个请求调用后端模型。

指标	基线	StepCache
平均延迟（秒）	2.13±0.04	0.67±0.04
中位延迟（秒）	2.42±0.05	0.01±0.00
P95 延迟（秒）	3.38±0.03	3.30±0.04
总令牌数（全部）	36.1±0.1k	27.3±0.9k
每个请求的词元数	162.7±0.6	123.0±4.0
质量通过率	72.5±0.5%	100%
最终检查通过率	72.5±0.5%	100%
结果分布（仅复用 / 修补 / 跳过）	–	79.7±2.4% / 5.4% / 14.9±2.4%

表 1：仅使用 CPU 的微基准测试摘要，取三个随机种子（42–44）的平均值。每个随机种子使用 222 个评估请求（每项任务 n=10 个基础提示，每次扰动 k=3 个变体）。StepCache 将平均延迟降低了 ∼3.2×，并将总令牌用量降低了 ∼24%，同时在任务专用检查和拼接输出完整性检查中，将正确性从 ∼72.5% 提高到 100%。

• **StepCache：** 预热阶段会为缓存填充数据；评估阶段使用检索、验证和选择性修补，并在触发时采用自适应跳过复用回退机制 + + 。

基准测试协议。 我们基于 n个基础提示构建了一个扰动密集型微基准测试（例如，一个数学模板和一个 JSON 模板），并针对每个基础提示，按不同扰动类型（例如低/中/高强度扰动和约束变更）生成 k 个扰动变体。我们采用两阶段协议：首先进入预热阶段，强制生成内容，为每个基础模板填充缓存；随后进入评估阶段，衡量扰动请求的复用、修补、跳过复用、令牌用量和延迟。在主要实验中，我们为每个任务使用 n=10 个基础提示，并为每种扰动使用 k=3 个变体；每个随机种子由此产生 222 个评估请求（随机种子 42–44）。我们报告平均值/中位数/P95 延迟、令牌用量、结果分布（仅复用/修补/跳过），以及在任务专用检查和拼接输出完整性检查下的正确性。-

5.2 指标

我们报告复用率、修补率和跳过复用率；后端调用次数；令牌用量；以及延迟统计（平均值/中位数/P95）。正确性通过两项检查进行衡量：(i) 任务专用质量检查（数学解答一致性；JSON 解析和必需键）以及 (ii) 对最终响应执行的拼接输出完整性检查，并记录失败原因（以检测步骤级检查产生的假阳性）。

5.3 结果

表 1 报告了每个随机种子下 222 个评估请求的种子平均结果（随机种子 42–44）。StepCache 将平均延迟从 2.13 秒降至 0.67 秒，同时在两项检查下均保持完全正确（质量检查通过率为 100%，最终检查通过率为 100%）。中位延迟降至近乎为零（0.01 秒），因为大多数请求都走仅复用快速路径；尾延迟仍主要受慢速路径（修补或跳过复用）影响，因此 P95 延迟仅略有变化。总体令牌用量减少了 ∼24%（36.1k→27.3k），这反映出大多数请求无需完全重新生成，且修补提示保持精简。

表 2 按任务和扰动细分了结果。键变更设置会强制重新生成结构化 JSON 步骤（100% 修补），但仍能保持正确性，且额外开销适中。对于数学任务，语义变更设置（值变更）会按设计触发跳过复用，从而避免在方程常数发生变化时进行无效复用。对于线性方程，StepCache 还通过有界修复尝试以及检查失败时的确定性回退机制来保证正确性。

6 讨论

StepCache 的优势所在。 当不同请求之间的解答结构稳定，且扰动仅局限于局部（例如改述、格式约束或小幅模式编辑）时，StepCache 的收益最大。在这些情况下，验证通过，系统便会执行快速路径（复用），从而大幅降低延迟和令牌用量。

Task	扰动	仅复用率	修补率	跳过率	节省的词元	最终率
json	low	100.0	0.0	0.0	119	100
json	med	100.0	0.0	0.0	84	100
json	high	100.0	0.0	0.0	89	100
json	键变化	0.0	100.0	0.0	-42	100
math	low	96.7	0.0	3.3	-17	100
math	med	96.7	0.0	3.3	19	100
math	high	96.7	0.0	3.3	18	100
math	值变更	0.0	0.0	100.0	-2	100

表 2：随机种子 42–44 下各任务的平均细分结果。仅复用/修补/跳过是互斥的每请求结果。键变化会触发针对 JSON 的结构化修补。值变更被视为语义变化情形，并触发跳过复用。节省的词元是每请求令牌用量差值（基线减去 StepCache）的平均值，四舍五入到最接近的整数。

语义变化与自适应策略。当扰动改变了核心语义（例如更改方程中的常数）时，朴素复用可能需要修补大部分步骤，其效率或许不如完全重新生成。StepCache 通过由不一致信号驱动的自适应跳过复用策略，明确应对这种情况：以少量后备后端调用为代价，确保正确性稳定并改善尾部行为。

完整性考量。任何缓存层都可能引入完整性风险（例如，在对抗性相似度下过度激进地复用、缓存投毒，或在存在未建模约束时复用过时步骤）。StepCache 通过任务感知验证、保守的重新生成策略和拼接输出验证来降低这些风险。采用更强的验证器（例如 LLM 评审或形式化检查器）、访问控制和缓存加固，是未来工作的重要方向。

7 结论

我们推出了 **StepCache**，这是一种用于大语言模型服务、与后端无关的缓存层，其运作方式是复用步骤级内容，而非整个响应或模型内部的 KV 状态。StepCache 会检索最匹配的先前请求，使用轻量级任务感知检查验证缓存步骤，并通过连续区块修补，仅重新生成未通过验证的区域；对于结构化输出，它采用有界修复循环；当预测复用无效时，则保守地回退为跳过复用。

在一项针对数学和 JSON 任务、包含大量扰动变体的仅使用 CPU 的微基准测试中（每个随机种子对应 222 个评估请求；n=10, k=3），对三个随机种子（42–44）取平均后，StepCache 将平均延迟从 2.13 秒降至 0.67 秒（中位数：从 2.42 秒降至 0.01 秒；第95百分位数：从 3.38 秒降至 3.30 秒），并将令牌总用量从 36.1k 降至 27.3k（每个请求 162.7→123 个令牌）；同时，在任务专用检查和拼接输出完整性检查下，正确性均从 72.5% 提升至 100%。在所有请求中，79.7% 走仅复用快速路径，5.4% 需要修补，14.9% 则因语义变化或不一致信号而触发跳过复用。

未来的工作包括：(i) 在生产级服务引擎（例如 vLLM）前端进行 GPU 支持的评估，并测量吞吐量和利用率指标；(ii) 在真实的突发流量轨迹下进行评估（例如采用 BurstGPT 风格的回放）；(iii) 将验证器从数学和 JSON 扩展到代码及工具增强型智能体工作流；(iv) 加强缓存完整性，以抵御对抗性相似度和投毒。

可复现性

我们的原型经过检测，可记录每个请求的决策和成本，包括复用/修补/跳过复用的结果、后端调用类型、令牌用量和延迟细分数据。每次运行都会生成：(i) 一份机器可读的基准测试 results.json，其中包含每个请求的记录和汇总统计数据（平均值/中位数/第95百分位数）；以及 (ii) 一份基准测试 mismatches.json 文件，记录任务级检查与拼接输出检查结果不一致的情况（包括失败原因），以便有针对性地调试假阳性/假阴性。

要复现报告的结果，请使用固定的种子和所需的缓存配置（例如，验证修补模式）运行基准测试脚本，并禁用可选的代码任务： -

python scripts/基准测试_扰动.py -n 10 -k 3 --seed 42 --include-code 0 python scripts/基准测试_扰动.py -n 10 -k 3 --seed 43 --include-code 0 python scripts/基准测试_扰动.py -n 10 -k 3 --seed 44 --include-code 0

[12] [13] [14]

参考文献

[1] Woosuk Kwon、Zhuohan Li、Siyuan Zhuang、Ying Sheng、Lianmin Zheng、Cody Hao Yu、Joseph E. Gonzalez、Hao Zhang 和 Ion Stoica。《利用 PagedAttention 高效管理大语言模型服务的内存》。*arXiv:2309.06180*，2023年。[2] Yuxin Wang、Yuhan Chen、Zeyu Li、Xueze Kang、Zhenheng Tang、Xin He、Rui Guo、Xin Wang、Qiang Wang、Amelie C. Zhou 和 Xiaowen Chu。《BurstGPT：用于优化大语言模型服务系统的真实工作负载数据集》。*arXiv:2401.17644*，2024年。[3] Shaoting Feng、Yuhan Liu、Hanchen Li、Xiaokun Chen、Samuel Shen、Kuntai Du、Zhuohan Gu、Rui Zhang、Yuyang Huang、Yihua Cheng、Jiayi Yao、Qizheng Zhang、Ganesh Ananthanarayanan 和 Junchen Jiang。《EVICPRESS：面向高效大语言模型服务的 KV 缓存压缩与驱逐联合优化》。*arXiv:2512.14946*，2025年。[4] Xinyue Ma、Heelim Hong、Taegeon Um、Jongseop Lee、Seoyeong Choy、Woo-Yeon Lee 和 Myeongjae Jeon。《OR-BITFLOW：面向服务级目标的长上下文大语言模型服务与细粒度 KV缓存重配置》。*arXiv:2601.10729*，2026年。[5] Bang Fu 和 Di Feng。《GPTCache：用于大语言模型应用的开源语义缓存，可加快响应并节省成本》。载于自然语言处理开源软件第三届研讨会论文集（*NLP-OSS 2023*），第212–218页，2023年。[6] Luis Gaspar Schroeder、Aditya Desai、Alejandro Cuadron、Kyle Chu、Shu Liu、Mark Zhao、Stephan Krusche、Alfons Kemper、Ion Stoica、Matei Zaharia 和 Joseph E. Gonzalez。《vCache：经验证的语义提示缓存》。*arXiv:2502.03771*，2025年。[7] Asmit Kumar Singh、Haozhe Wang、Laxmi Naga Santosh Attaluri、Tak Chiam 和 Weihua Zhu。《面向分层大语言模型架构的异步经验证语义缓存》。*arXiv:2602.13165*，2026年。[8] Nils Reimers 和 Iryna Gurevych。《Sentence-BERT：使用孪生BERT网络的句子嵌入》。*arXiv:1908.10084*，2019年。[9] Wei Wang、Furu Wei、Li Dong、Hangbo Bao、Nan Yang 和 Ming Zhou。《MiniLM：面向任务无关压缩预训练Transformer的深度自注意力蒸馏》。*arXiv:2002.10957*，2020年。[10] Jeff Johnson、Matthijs Douze 和 Herv´e J´egou。《利用图形处理器进行十亿规模的相似性搜索》。*arXiv:1702.08734*，2017年。[11] An Yang 等。《Qwen2.5 技术报告》。*arXiv:2412.15115*，2024年。[12] Azam Nouri。《利用平面曲率和梯度方向进行手写识别的多层感知机基线》。《国际计算机应用期刊》，第187卷，第47期，第1–5页，2025年。doi：10.5120/ijca2025925791。[13] Azam Nouri。《用于手写字符识别的 Sobel梯度多层感知机基线》。*arXiv* 预印本 *arXiv:2508.11902*，2025年。可访问：https://arxiv.org/abs/2508.11902。[14] Azam Nouri。《面向大语言模型的显著性增益对编码：基于频率的子词合并的统计替代方案》。*arXiv* 预印本 *arXiv:2603.19261*，2026年。doi：10.48550/arXiv.2603.19261。