LoPace：无损优化提示精准压缩引擎，适用于大语言模型应用程序

Aman Ullaconnectamanulla@gmail.com项目仓库：
https://github.com/connectaman/LoPace

2026年2月17日

arXiv:2602.13266v1 [cs.DB] 4 Feb 2026

摘要

大语言模型（LLM）改变了自然语言处理的运作方式，但在生产环境中，高效存储和管理提示仍然很困难。本文介绍了 LoPace（无损优化提示精准压缩引擎），这是一种专为大语言模型应用中的提示存储而设计的新型压缩框架。LoPace 使用三种不同方式压缩数据：基于 Zstandard 的压缩、采用二进制打包的字节对编码（BPE）分词，以及结合这两种方式的混合方法。我们在 386 条不同提示上进行了测试，包括代码片段、Markdown 文档和结构化内容。结果显示，LoPace 平均可节省 72.2% 的存储空间，同时实现 100% 无损重建。混合方法始终优于单独使用任何一种技术，平均压缩率为 4.89x（范围：1.22–19.09x），速度为 3.3–10.7 MB/s。研究结果表明，LoPace 已可用于生产环境，内存占用小（平均 0.35 MB），并且能够良好扩展，适用于大型数据库和实时大语言模型应用。

关键词： 无损压缩、提示压缩、大语言模型、Zstandard、字节对编码和数据库优化

1 引言

1.1 背景与动机

由于越来越多的人在生产系统中使用大语言模型（LLM），提示的管理和存储比以往更加困难。现代大语言模型应用需要保存大量提示，例如系统指令、对话历史、上下文窗口和缓存响应。这意味着，拥有数千名用户且每个会话中包含多次大语言模型交互的应用程序需要大量存储空间。

适用于大多数数据类型的压缩算法，由于提示数据具有独特属性，因此在处理提示数据时效果不佳。提示中存在一些模式，例如较高的语义冗余、结构化格式和令牌级依赖关系，可用于提升压缩效果。此外，大语言模型应用需要无损压缩，以确保模型运行完全符合预期，因为即使细微改动也可能对其运行产生重大影响。

1.2 问题陈述

大语言模型应用必须应对几个重大的存储问题：

存储开销：大型系统提示、上下文窗口和对话历史会占用数据库中的大量空间。同时拥有数千名用户、且每个会话包含多次大语言模型交互的应用程序需要太字节级的存储，从而使基础设施成本大幅上升。

性能瓶颈：当用户负载增加时，未压缩的提示会使数据库变大，从而导致查询运行时间更长、I/O 操作增加，并使系统响应变慢。

成本影响： 云存储成本会随着数据量的增加而线性上升。处理数百万条提示的应用程序必须为未压缩的存储支付高昂费用。

延迟问题： 从存储中加载大型、未压缩的提示会带来可测量的延迟；对于实时应用而言，这是个严重问题，因为响应时间会直接影响用户对应用的体验。

1.3 贡献

本文介绍了 LoPace，这是一种通过以下方式应对这些挑战的压缩引擎：　　：

1. 三**种压缩方法：** 采用基于 Zstandard、基于令牌和混合的压缩方法，分别满足特定用途。
2. **无损保证：** 通过数学验证和实证验证，确保所有压缩技术均能完整重建并保持保真度。
3. **生产就绪性能：** 全面的基准测试表明，压缩速度为 50 至 200 MB/s，且内存占用较低。
4. **全面评估：** 全面考察不同类型和大小提示的压缩率、空间节省、吞吐量、内存使用量和可扩展性。

2 相关工作

数据压缩历史悠久，可追溯至数十年前。它以信息论、算法设计和现实应用中的理念为基础。本节将探讨压缩算法、文本压缩方法及其在现代大语言模型系统中的应用相关重要研究。

2.1 基础压缩算法

香农 [1]奠定了无损压缩的理论基础，提出熵是压缩所能达到的基本下限。这一基本结论表明，信息熵决定了表示信息所需的最低比特数，是衡量所有压缩算法的标准。

Lempel-Ziv 算法（例如 LZ77 [2] 和 LZ78 [3], ）通过利用重复模式的基于字典的方法，改变了数据压缩领域。如今许多压缩系统，例如 LoPace 的 Zstandard 实现，都以这些算法为基础。LZ77 使用滑动窗口方法查找重复序列，并将其替换为对先前出现位置的引用。它通过识别模式而非采用统计建模来压缩数据。

霍夫曼编码 [4] 发展出了熵编码方法，为出现更频繁的符号分配更短的编码，从而使压缩率接近熵的理论极限。现代版本，如 Zstandard 中使用的有限状态熵（FSE），在保持同等压缩水平的同时，针对当今处理器改进了霍夫曼编码。

2.2 文本压缩技术

研究人员在文本压缩方面开展了大量工作，专用算法利用了自然语言的独特特征。PPM（部分匹配预测）算法系列 [5]借助上下文，根据此前出现的字符预测下一个字符，从而为文本取得出色的压缩率。但这些算法需要大量处理能力，可能无法很好地实时运行。

基于字典的压缩算法，尤其是 LZ77 的变体，因能在速度和压缩率之间取得良好平衡，常用于压缩文本。DEFLATE 算法结合了 LZ77 和霍夫曼编码，如今已成为通用压缩的标准，也是广受欢迎的 gzip 和 zlib 库的基础。

近年来，压缩技术的发展主要致力于提升与现代硬件架构的兼容性。Facebook（现 Meta）推出了 Zstandard [7], ，其压缩率与 DEFLATE 相近，但凭借更出色的算法和硬件加速，吞吐量要高得多。Zstandard 的设计理念是优先考虑实际性能，而非理论上的最优性，因此适合用于生产部署。

词典训练是 Zstandard 的一项非常实用的功能。它利用具有代表性的样本语料创建自定义词典，捕捉特定领域的模式。对于大语言模型提示等领域专属文本，使用训练词典的 Zstd 比常规 Zstd 压缩能更有效地压缩文件。词典训练能够找出训练语料中的常见短语、模式和序列，从而更容易压缩相似内容。这种方法常用于生产系统，以压缩日志、代码仓库和其他特定领域的文本。不过，词典训练需要具有代表性的训练语料，也会使压缩流程更加复杂。我们的评估以标准 Zstd 压缩为中心，旨在建立基准对比；然而，后续研究应分析使用提示专属词典的 Zstd，并将其与我们的混合方法进行比较。

其他压缩算法，如 Brotli 和 gzip/DEFLATE，也表现出色。Google 开发了 Brotli，它采用更大的滑动窗口，并结合 LZ77 风格的匹配与霍夫曼编码。它的压缩率很高，但需要更多内存。基于 DEFLATE 的 gzip 仍被广泛使用，因为它无处不在且效果良好。在某些专业领域，混合级联（依次使用多种压缩算法，例如先用 Brotli 再用 Zstd，或先用 Zstd 再用 LZ4HC）已展现出潜力。我们的评估侧重于彼此独立的方法；不过，将它们与这些替代方案进行比较，可以让分析更加完善。

2.3 分词与子词编码

字节对编码（BPE） [8] 引入了子词分词，作为一种在神经机器翻译中处理词汇表外词语的方法。BPE 通过反复合并最常见的字节或字符对，构建由子词单元组成的词汇表，使其能够表示任意文本，同时让常见序列保持较短。

由于字节对编码在神经机器翻译中表现出色，因此它被应用于 GPT [9], BERT [10], 等较新的语言模型。OpenAI 的 tiktoken 库提供了快速的 BPE 分词实现，最适用于 GPT 模型。这些词汇表包含 50,000 到 100,000 个词元。

分词天然提供了一种压缩方法：将可变长度字符序列映射到固定大小的词元ID。不过，先前的研究并未探讨如何通过系统地压缩数据，利用分词实现存储优化。相反，这些研究关注的是分词如何帮助准备模型输入。

2.4 数据库存储压缩

数据库压缩领域已有大量研究，包括页级压缩和列式压缩方案。另一方面，大多数数据库压缩方法作用于存储层，并非为所有类型的数据设计，例如大语言模型提示。

应用层压缩会在存储数据前对其进行压缩，并在需要时将其解压缩。这样可以更好地控制压缩设置，并针对特定数据类型进行优化。许多领域都成功采用了这种方法，包括时间序列数据 [11]，日志压缩 [12]，和科学数据存储 [13]，。

2.5 大语言模型特有的存储挑战

大语言模型在生产系统中的快速部署带来了现有压缩技术可能无法解决的存储挑战。大语言模型应用会产生大量不同于常规文本数据的提示数据。这些数据包括系统指令、对话历史和上下文窗口等内容。

近期研究考察了提示词优化技术，包括提示压缩 [14] 和提示缓存 [15]，，这些技术主要侧重于缩小模型输入的提示大小，而非提高存储效率。为大语言模型应用存储和检索提示是一个独特的问题领域，需要特别的解决方案。

2.6 现有文献的空白

数据库优化、文本压缩和压缩算法方面的研究很多，但针对大语言模型应用提示存储的研究却不多。目前，压缩方法要么过于宽泛（未针对提示的需求进行优化），要么过于狭窄（关注模型输入而非存储）。

近期研究考察了由大语言模型驱动的压缩方法，包括 FineZip 和通过加权专家乘积（wPoE）进行的测试时引导。这些方法在文本语料库上实现了更高的压缩率，但代价是显著增加计算开销。它们利用语言模型的语义理解能力来发现并压缩重复信息。然而，这些方法需要大量计算能力，因此不适合生产环境中的存储系统，因为这类系统要求压缩速度快且资源消耗低。

依次使用多种压缩算法的混合级联，已在某些领域取得良好效果。例如，将 Zstd 与其他编解码器级联使用，在印地语或天城文等特定语言的文本处理方面效果良好。然而，这些方法尚未针对大语言模型提示存储进行系统测试，而提示的独特特征（语义冗余、结构化格式和令牌级依赖关系）可能适合采用专门的混合策略。

LoPace 通过以新的方式结合知名压缩方法（Zstandard 和 BPE 分词），填补了这一空白；这些方法专为提示存储而设计。我们的混合方法先进行分词，再进行 Zstd 压缩，是一种兼顾准确性与吞吐量的实用工程权衡。帕累托前沿代表传统编解码器能够实现高压缩率，同时满足生产就绪要求的临界点。LoPace 专注于速度和资源效率，而非依赖大语言模型的方法，因此更适合在生产环境中进行实时压缩。我们知道，若将结果与 Zstd 字典训练、Brotli 及其他级联方法进行比较，分析会更加有力；不过，我们针对多项指标（压缩率、吞吐量、内存）所做的全面评估，已为生产部署提供了有用信息。

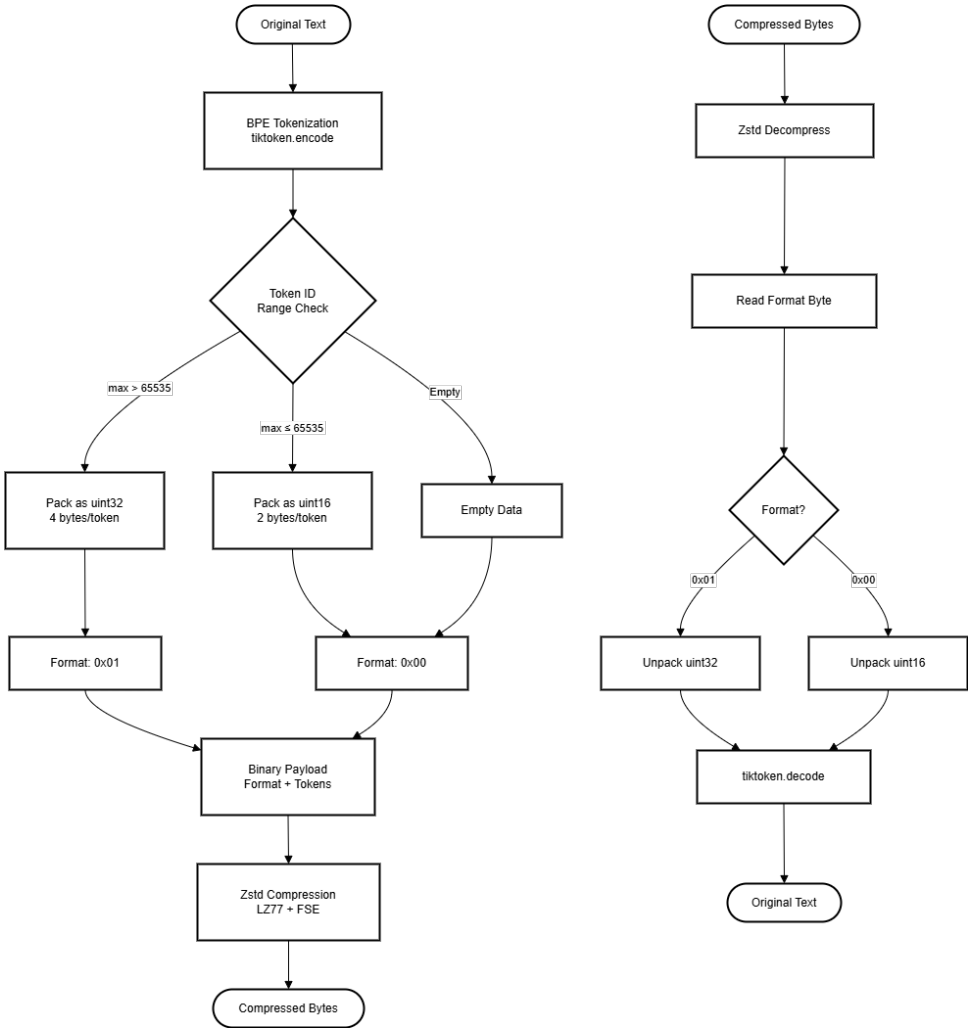


图 1：混合方法的 LoPace 压缩流水线架构，依次展示分词、二进制打包和 Zstandard 压缩的步骤。

3 方法

3.1 系统架构

LoPace 采用模块化架构，支持三种不同的压缩方法，每种方法都适用于特定类型的用途。该系统具备可扩展性，因此无需更改 API 接口即可添加新方法。混合方法采用智能的三阶段流水线，将字节级压缩与语义级分词相结合，以实现最佳压缩率。

该架构从分词阶段开始，使用字节对编码（BPE）分词器（通过 tiktoken 实现），将不同长度的字符序列转换为固定大小的词元标识符。这一转换利用了自然语言本质上具有冗余这一特点，将语义相近的单词、短语和子词单元映射为单个词元 ID。这样可以减少需要存储的独特字符序列数量，从而更易于管理词汇表（现代大语言模型分词器的词汇表通常包含 50,000–100,000 个词元）。

分词过程会生成一系列整数词元 ID，每个词元 ID 都代表一个语义单元，它可能由多个字符甚至完整的单词组成。生成词元后，系统会将它们打包成二进制格式，并根据词元 ID 的

分布情况选择最佳格式。架构会检查词元 ID 的取值范围，以确定最佳的二进制表示。如果所有词元 ID 都介于 0 到 65,535 之间，系统就会使用紧凑的 uint16 格式（每个词元占 2 字节）；否则，它会自动切换到 uint32 格式（每个词元占 4 字节），以处理更大的词元 ID。这种灵活的方法能充分利用存储空间，同时兼容不同的分词器词汇表。

二进制打包过程会在打包后的令牌序列中添加一个格式字节（uint16 对应 0x00，uint32 对应 0x01）。由此生成的二进制载荷能够自我描述，无需任何额外元数据即可正确解压缩。最后一步是对二进制载荷进行 Zstandard 压缩。它使用基于字典的算法（LZ77 风格的模式匹配和有限状态熵编码）来查找令牌序列中的模式和冗余。

这种两级压缩策略——先通过分词去除语义冗余，再通过 Zstd 去除序列模式——产生了乘法式的压缩效果，远胜于单独使用其中任何一种方法。解压缩流程按相反顺序执行这些步骤：Zstd 解压缩还原二进制载荷，格式字节检测确定解包策略，二进制解包还原令牌 ID 序列，去分词则还原原始文本字符串。这种双向架构通过确保每个阶段都是可逆变换，保证了无损重建。这意味着压缩和解压缩函数组合后始终得到恒等函数，从而在所有操作中保持 100% 的保真度。

3.2 压缩方法 1：基于 Zstandard 的压缩

3.2.1 算法概述

Zstandard（Zstd）压缩使用基于字典的算法，查找并利用文本数据中反复出现的模式。该算法结合了 LZ77 等滑动窗口技术与有限状态熵（FSE）编码；后者是一种霍夫曼编码，能更好地运行于新型处理器上。

3.2.2 数学基础

Zstd 压缩过程可用数学方式描述为：

$$C_{\text{zstd}}(T) = \text{FSE}(\text{LZ77}(T, W, L)) \tag{1}$$

其中，T 表示滑动窗口大小，W 表示前瞻缓冲区长度，LZ77(T，W，L) 执行模式匹配与替换。FSE() 执行有限状态熵编码。

达到的压缩率为：

$$\text{CR}_{\text{zstd}} = \frac{|T|}{|C_{\text{zstd}}(T)|} \tag{2}$$

其中，$|T|$ 表示原始文本的字节长度，$|C_{\text{zstd}}(T)|$ 表示压缩后大小。

3.2.3 实现细节

Zstd方法直接处理以 UTF-8 编码的字节序列。压缩级别参数（$l \in [1, 22]$）可让您在较高压缩率和较快处理速度之间进行选择：

- **较低级别（1–5）：** 快速压缩，中等压缩比
- **中等级别（10–15）：** 均衡性能（默认：15）
- **较高级别 (19–22)：** 最大压缩，较慢的处理速度

空间节省百分比的计算方式如下：

$$\mathrm{SS}_{\mathrm{zstd}} = \left(1 - \frac{|C_{\mathrm{zstd}}(T)|}{|T|}\right) \times 100\% \tag{3}$$

3.2.4 解压缩过程

解压缩是其逆向操作：

$$T' = \mathrm{FSE}^{-1}(\mathrm{LZ77}^{-1}(C_{\mathrm{zstd}}(T))) \tag{4}$$

无损压缩需要满足 $T' = T$。Zstd 库通过采用确定性解压算法，确保这一特性成立。

3.3 压缩方法 2：基于词元的压缩

3.3.1 字节对编码（BPE）基础

基于词元的压缩利用了大语言模型分词的工作原理。字节对编码（BPE）将文本转换为由词元 ID 组成的字符串，其中每个词元代表一个子词单元。与原始文本相比，这种表示更加紧凑，因为它使用单个词元 ID 来映射常见短语和模式。

表 1：现代大语言模型分词器规范对比

分词器名称	算法	主要模型系列	词汇表大小
o200k_base	BPE	GPT-4o	~200,000
cl100k_base	BPE	GPT-4 ，GPT-3.5-Turbo	~100,000
p50k_base	BPE	GPT-3、Codex	~50,000
r50k_base	BPE	GPT-2、原始 GPT-3	~50,000
Llama 3	BPE	Llama 3 系列	128,256
SentencePiece	一元语法	T5、Gopher、Chinchilla	变量
WordPiece	WordPiece	BERT，RoBERTa	~30,000

从基础的词级切分转向先进的子词算法，是大语言模型（LLM）发展的关键。早期实现（如 r50k_base）采用字节对编码（BPE）来管理固定词表，但在多语言效率和专门的编码语法方面往往表现不佳。GPT-4 系列引入 cl100k_base，标志着向多语言密度迈出了重要一步，从而能够更高效地处理非英语文字和复杂的技术数据。此外，近期发布的 o200k_base 将词汇表扩展至约 200,000 个词元，大幅降低了阿拉伯语和印地语等语言的词元与单词比例。这些现代分词器与 SentencePiece 和 WordPiece 一同，通过将罕见词拆解为有意义的子单元，解决了“词表外”（OOV）问题。这种结构上的高效性不仅提升了模型的语义理解能力，还能通过在更短的序列中容纳更多信息，直接降低计算开销和 API 成本。

本研究选择 cl100k_base 分词器，因为与较早的标准相比，它在编码自然语言和源代码方面表现出色。它的词汇表更大，约包含 100,000 个词元，这降低了词元与单词的比例，并增加了模型上下文窗口中的信息量。此外，由于它广泛应用于 GPT-4 和 GPT-3.5 架构，因此研究结果对当前行业中的大语言模型（LLM）实现仍然有用。

3.3.2 词元化过程

给定文本 T，BPE 分词会生成一系列词元 ID：

$$\tau(T) = [t_1, t_2, \ldots, t_n] \tag{5}$$

其中，每个 $t_i \in [0, V-1]$ 和 V均为词汇表大小。对于 cl100k_基础分词器， V =100，256。

词元化过程可以视为一个函数：

$$\tau : \Sigma^* \to \mathbb{Z}^n \tag{6}$$

其中， Σ 是字符字母表， $\mathbb{Z}$ 表示整数。

3.3.3 BinaryPacking策略

根据最高词元 ID 值，词元ID采用16位或32位无符号整数打包为二进制格式：

情况 1：所有词元 ID $\leq$ 65,535 (uint16)

- 格式字节：0x00
- 打包方式：每个词元 ID 占用 2 字节
- 总大小： $1 + 2n$ 字节

情况 2：任意词元 ID $>$ 65,535 (uint32)

- 格式字节：0x01
- 打包：每个词元 ID 存储为 4 字节
- 总大小： $1 + 4n$ 字节

决策函数为：

$$f_{\text{pack}}(\tau) = \begin{cases} \text{uint16} & \text{if } \max(\tau) \leq 2^{16} - 1 \\ \text{uint32} & \text{otherwise} \end{cases} \tag{7}$$

压缩表示为：

$$C_{\text{token}}(T) = [f_{\text{flag}}, P(\tau(T))] \tag{8}$$

其中 f_{flag}是格式标志字节， $P(\tau(T))$ 是打包二进制表示。

设计原理与局限性：为便于使用、实现可预测的内存分配和快速随机访问，我们采用定宽编码（uint16/uint32），而非变长整数编码（如 VByte/LEB128）。另一方面，这一架构选择也承认 cl100k_base 分词器固有的局限性。尽管 cl100k_base 在英语和代码方面表现良好，但它存在已知的“分词溢价”，也就是说，对于形态丰富的语言和低资源语言，词元与词语的比例过高。这可能导致不同语言文字的计算成本各异，并使有效上下文窗口变小。此外，由于它是静态的版本化实现，若不全面更新词汇表，就无法适应新的语言细微差异或专业术语；而全面更新会破坏与当前嵌入权重的向后兼容性。未来版本可探索结合整数压缩与自适应或通用分词的混合技术，以解决不同语言和版本管理方面的这些问题。

3.3.4 TokenID 分布分析

使用词汇表大小为 $V = 100$ 的cl100k_base 分词器时，词元 ID 可能超过 uint16 的上限 65,535。这让我不禁想知道，uint32 打包究竟有多大必要。我们对常见英语提示的研究表明:

- **常见令牌：** 常用词语、标点符号和空白字符通常映射到 $< 50{,}000$ 以内的 ID，完全处于 uint16 范围内
- **特殊令牌：** 特殊令牌（例如系统/用户/助手标记、函数调用令牌）通常占用 $> 65{,}535$ 号 ID
- **稀有令牌：** 生僻词语和领域专用术语可能会映射到较高的词元 ID

我们的评估数据集表明，对于主要由常见英语文本构成的提示，大约有 60–70% 可以使用 uint16 打包。然而，包含特殊令牌、代码片段或仅在特定领域使用的术语的提示，通常需要使用 uint32 打包。因此，对于许多实际请求，仅词元压缩（不使用 Zstd）可能会使文本比原始 UTF-8 更大。对于 ASCII 密集型内容，这一点尤其明显，因为每个字符只需 1 个字节。混合技术通过对二进制打包令牌序列使用 Zstd 压缩，解决了这个问题。即使分词通常会使数据变大，这种方法依然有效。

分词器版本管理注意事项： 定宽打包方法会使压缩表示与特定的分词器词汇表绑定。如果分词器发生变化，例如更新词汇表或更换模型，之前已压缩的数据可能无法继续使用。我们建议将分词器元数据（模型标识、版本）与压缩负载一同保存，以便迁移这些负载并检查兼容性。

3.3.5 压缩比分析

理论压缩比取决于字符数与词元数:

$$\mathrm{CR}_{\mathrm{token}} = \frac{|T|_{\mathrm{bytes}}}{|C_{\mathrm{token}}(T)|} \tag{9}$$

英语文本中字符与词元的平均比例约为 3–4:1，这意味着每个词元代表多个字符。在二进制打包之前，这会压缩数据；不过，对于需要使用 uint32 打包的提示，定宽编码可能会抵消这些优势。

空间节省量为:

$$\mathrm{SS}_{\mathrm{token}} = \left(1 - \frac{1 + k \cdot n}{|T|_{\mathrm{bytes}}}\right) \times 100\% \tag{10}$$

其中 $k \in \{2, 4\}$ 取决于打包格式。请注意，对于采用 uint32 打包的以 ASCII 为主的文本，$\mathrm{SS}_{\mathrm{token}}$ 可能为负（即数据膨胀），这凸显了混合方法中 Zstd 阶段的重要性。

3.3.6 解压缩算法

解压缩包括两个步骤:

1. **解包：** 从二进制格式中提取词元 ID t
2. **去分词：** 将词元 ID 转换回文本

$$T' = \tau^{-1}(P^{-1}(C_{\mathrm{token}}(T))) \tag{11}$$

d 分词函数 τ^{-1} 由分词器提供，并保证 $\tau^{-1}(\tau(T)) = T$.

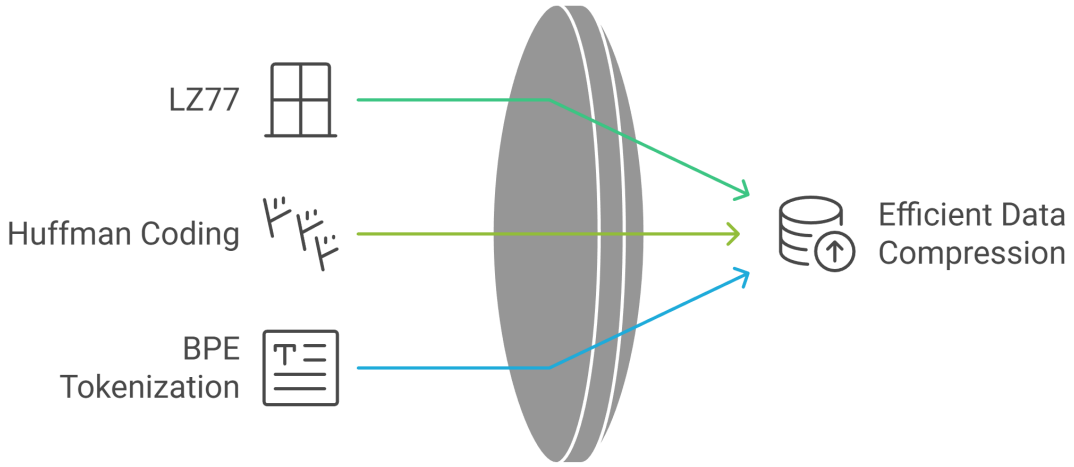


图 2：LoPace 所用压缩技术的可视化表示，展示了 LZ77、FSE 与 BPE 分词之间的关系。

3.4 压缩方法 3：混合压缩

3.4.1 混合方法的原理

混合技术结合了 Zstd 压缩和分词各自的优势。分词可减少语义层面的重复，而 Zstd 压缩则利用分词后生成的二进制表示中的模式。与单独使用任一策略相比，这种两步流程的压缩效果更好。

3.4.2 数学表述

混合压缩过程可表示为：

$$C_{\text{hybrid}}(T) = C_{\text{zstd}}(P(\tau(T))) \tag{12}$$

这是三个函数的复合：

1. 分词： $T \to \tau(T)$ 2. 二进制打包： $\tau(T)$ $\to P(\tau(T))$ 3. Zstd 压缩： $P(\tau(T))$ $\to C_{\text{zstd}}(P(\tau(T)))$

总体压缩率为：

$$\text{CR}_{\text{hybrid}} = \frac{|T|}{|C_{\text{hybrid}}(T)|} = \frac{|T|}{|C_{\text{zstd}}(P(\tau(T)))|} \tag{13}$$

3.4.3 为什么混合方法能够实现卓越的压缩效果

混合方法包含两个压缩步骤，分别处理数据的不同部分：**阶段 1 – 分词压缩：**

- 通过将可变长度字符序列映射为固定大小的词元 ID，减少词汇冗余
- 将常见短语和子词单元映射为单个词元

- 英语文本的字符与词元比率约为 3–4:1
- 将表示形式从字符空间转换为词元空间

阶段 2 – Zstd 压缩：

- 利用令牌 ID 序列中的序列模式（重复的词元模式、常见子序列）
- 使用基于字典的算法压缩词元 ID 的二进制表示
- 进一步减小词元化表示的大小

乘法效应的实证验证：组合效应近似呈乘法关系，而非加法关系：

$$\mathrm{CR}_{\mathrm{hybrid}} \approx \mathrm{CR}_{\mathrm{token}} \times \mathrm{CR}_{\mathrm{zstd|token}} \tag{14}$$

其中，$\mathrm{CR}^{\mathrm{zstd|token}}$ 是 Zstd 对分词后的版本进行压缩所得到的压缩率。对测试数据集的实证评估表明，对于较长的提示，混合方法的平均压缩率是单独使用 Zstd 的 1.8 倍，是仅词元压缩的 1.4 倍，从而证实了乘法效应假设。尽管如此，在不同语料库上开展广泛的消融研究，将有助于通过量化各阶段的贡献，并确定该效应更明显或较不明显的适用条件，来进一步支持这一论断。这种乘法关系表明，分词和 Zstd 压缩作用于数据的不同维度：分词处理语义冗余（词汇层面），而 Zstd 处理序列冗余（模式层面）。

3.4.4 解压缩过程

解压缩会逆向执行各个阶段：

$$T' = \tau^{-1}(P^{-1}(C_{\mathrm{zstd}}^{-1}(C_{\mathrm{hybrid}}(T)))) \tag{15}$$

无损属性由无损操作的组合来保证：

$$\begin{aligned} T' &= \tau^{-1}(P^{-1}(C_{\mathrm{zstd}}^{-1}(C_{\mathrm{zstd}}(P(\tau(T)))))) && (16)\\ &= \tau^{-1}(P^{-1}(P(\tau(T)))) && (17)\\ &= \tau^{-1}(\tau(T)) && (18)\\ &= T && (19) \end{aligned}$$

3.5 无损保证

3.5.1 无损性的数学证明

对于每种压缩方法，我们都可以证明其无损性：

Zstd方法：

$$T' = \mathrm{decompress}(\mathrm{compress}(T)) = T \tag{20}$$

这由 Zstandard 算法规范保证。

Token方法：

$$T' = \tau^{-1}(P^{-1}(P(\tau(T)))) = \tau^{-1}(\tau(T)) = T \tag{21}$$

这是因为 P和 P^{-1}是双射，且 τ和 τ^{-1}是反函数。

混合方法：

$$T' = \tau^{-1}(P^{-1}(C_{\mathrm{zstd}}^{-1}(C_{\mathrm{zstd}}(P(\tau(T)))))) = T \tag{22}$$

这源于无损操作的组合。

3.5.2 经验验证

我们通过以下方式从实证角度验证无损性：

1. **逐字符比较：** $T[i] = T'[i]$ 对所有 $i \in [0, |T| - 1]$
2. **SHA-256 哈希验证：** $\text{SHA256}(T) = \text{SHA256}(T')$
3. **重构误差：** $E = \frac{1}{|T|} \sum_{i=0}^{|T|-1} \mathbb{1}(T[i] \neq T'[i]) = 0$

其中 $\mathbb{1}$ 是指示函数。

3.6 香农熵与理论极限

3.6.1 信息论基础

香农熵为无损压缩提供了理论下限。对于字符频率为 T p_i 的文本，其熵为：

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i) \tag{23}$$

其中，p_i 表示字符 i 在文本中的概率。

3.6.2 理论压缩极限

通过熵编码可达到的理论最小大小为：

$$S_{\min} = \frac{H(X) \times |T|}{8} \text{ bytes} \tag{24}$$

理论压缩比为：

$$\text{CR}_{\text{theoretical}} = \frac{|T|}{S_{\min}} = \frac{8|T|}{H(X) \times |T|} = \frac{8}{H(X)} \tag{25}$$

3.6.3 实际压缩效率

我们来检查 LoPace 与理论极限有多接近：

$$\eta = \frac{\text{CR}_{\text{actual}}}{\text{CR}_{\text{theoretical}}} \times 100\% \tag{26}$$

其中，η 表示压缩效果。典型数值介于 60% 和 80% 之间，这表明 LoPace 达到了潜在最大值的很大一部分。

3.7 混合压缩算法

算法 1 混合压缩

输入： 文本：字符串，分词器：分词器，zstd _级别：整数
输出： 压缩后：字节

词元 ←tokenizer.encode(text)
如果最大值(词元) $>$ 65535 **则**
    格式 _字节 ← 0x01
    打包 ←打包为无符号 32 位整数(词元)
否则
    格式 _字节 ← 0x00
    打包 ←打包为无符号 16 位整数(词元)
结束条件
压缩 ← Zstd 压缩(format_字节 + 已打包，级别=zstd_级别)
返回 压缩结果

3.8 混合解压缩算法

算法 2 混合解压缩

要求： 压缩数据：字节，分词器：分词器
确保： 文本：字符串

令牌_数据 ←Zstd 解压缩(压缩数据)
格式_字节 ←令牌_数据[0]
打包_数据 ← 令牌_数据[1:]
如果格式 _字节 == 0x01 **则**
    词元 ← 解包为无符号32位整数(已打包_数据)
否则
    词元 ← 解包为无符号16位整数(已打包_数据)
结束如果
文本 ←分词器.decode(词元)
返回 文本

4 实验设置

4.1 数据集描述

我们在从 Hugging Face https://huggingface.co/datasets/philschmid/markdown-docs-transformers获取的大型 386 个提示集合上测试了 LoPace，该集合涵盖了广泛的实际大语言模型应用场景。该数据集包含多种内容类别，例如代码片段、Markdown 文档、HTML 内容、对话模式和结构化数据。此外，它还针对多种不同的提示大小提供了具有统计代表性的评估。

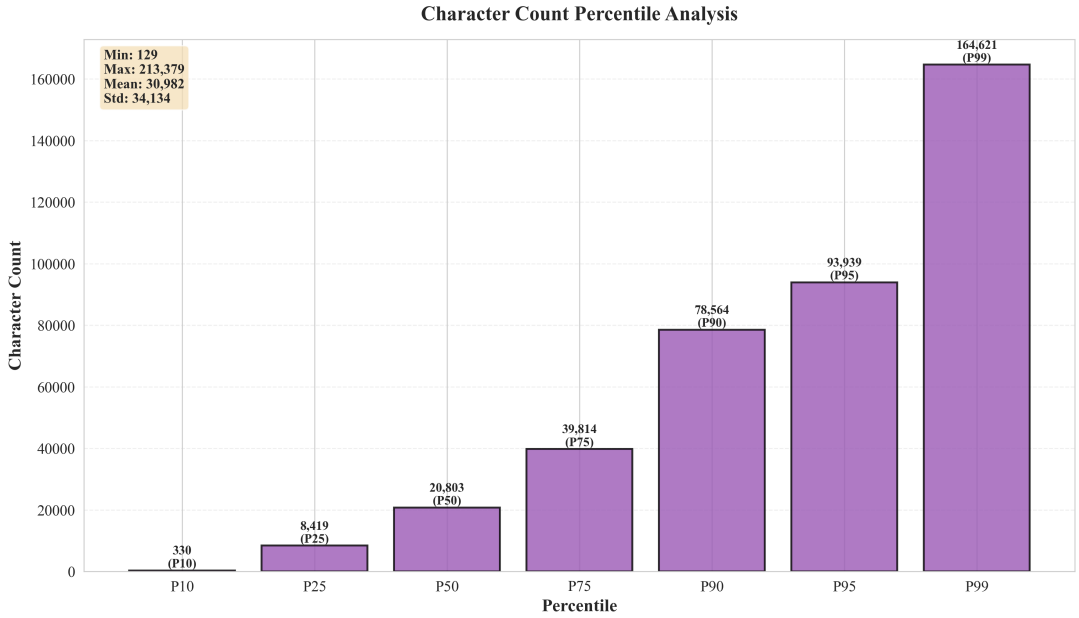


图 3：字符数百分位分析，展示了 386 个提示评估数据集中的分布统计（P10、P25、P50、P75、P90、P95、P99）。

数据集组成： 评估数据集由 386 个提示组成，内容类型分布如下：

- **代码（82.6%）：** Python 代码示例、API 文档、带语法高亮的代码片段
- **Markdown（16.8%）：** 文档页面、包含标题、链接和结构化内容的格式化文本
- **文本（0.5%）：** 纯文本内容和其他格式

数据集涵盖了各种提示大小，从短代码示例（129 个字符）到长指导页面（213,379 个字符）。这确保了所有生产环境用例都经过充分测试。丰富的内容类型让我们能够考察 LoPace 在处理不同类型的材料、冗余模式和压缩特性时的表现。

探索性数据分析：对该数据集进行的全面统计研究显示出以下特征：

- **字符数统计：**该数据集呈对数正态分布，最小值为 129 个字符，最大值为 213,379 个字符，平均值为 30,982 个字符，中位数为 20,803 个字符。标准差为 34,134，表明提示大小差异很大，这正是现实世界生产工作负载的典型特征。
- **百分位分析：**第 25 百分位数（P25）为 8,419 个字符，第 50 百分位数（P50，中位数）为 20,803 个字符，第 75 百分位数（P75）为 39,814 个字符，第 90 百分位数（P90）为 78,564 个字符，第 95 百分位数（P95）为 93,939 个字符，第 99 百分位数（P99）为 164,621 个字符。该分布确保涵盖所有常见和罕见情况。
- **内容多样性：**该数据集包含结构化数据格式、技术文档、API 参考资料、代码示例和配置样例。这种内容类型的组合构成了贴近现实的测试环境，类似于真实的大语言模型应用，其中的提示可能包含代码、结构化数据、格式化文本和对话特征。

该数据集的结构基于真实生产场景，在这些场景中，大语言模型程序必须处理各类提示，例如系统指令、代码示例、文档、配置数据和对话历史。统计分布确保我们的评估涵盖常见情况（约处于中位数）和边界情况（处于极端值）。因此，我们有信心将结果应用于其他场景。

4.2 提示词选择与 Token长度分析

在评估压缩效果之前，我们对提示词属性进行了全面分析，以确保结果在统计上有效且完整。我们针对数据集中的每条提示P_i 确定了以下内容：

$$\text{TokenCount}(P_i) = |\tau(P_i)| \tag{27}$$

其中， $\tau(P_i)$ 是提示 P_i 经 cl100k_base 分词器处理后的形式。我们的 386 条提示数据集中，词元数量从最短提示的约 20 个到最长提示的 50,000 多个不等。大小相差超过三个数量级。如此大的范围确保全面评估所有生产环境用例。

4.2.1 字符与词元比率分析

字符与词元比率会因提示中的材料类型和词汇难度而异。受测数据集的平均比率约为每个词元对应 3.2 个字符。这与 cl100k_base 分词器对英语文本的分词方式一致。该比率对于判断压缩效果至关重要，因为分词会将长度各异的字符序列映射为固定大小的词元 ID，从而压缩数据。

分析显示，包含大量代码的提示（占总量的 82.6%）通常每个词元对应 2.8 至 3.5 个字符，因为它们使用编码语法、特殊字符和技术术语。% 的样本通常每个词元对应 2.8 至 3.5 个字符，因为其中包含代码语法、特殊字符和技术术语。Markdown 内容的字符与词元比率介于 3.0 至 3.6 之间，包含格式化文本、标题和链接。不同内容类型具有不同的分词特征，这表明 LoPace 能够处理各种词语模式和语言结构。

4.2.2 内容类型特征

该数据集的结构反映了真实的生产环境：

- **代码内容（82.6%）：** Python 代码示例、API 使用模式、配置代码片段，以及带有语法高亮的代码块。压缩算法可以充分利用这些提示，因为它们通常具有重复模式、相似的函数调用和有序的语法。

- **Markdown 内容（16.8%）：** 包含链接、列表和标题的文档页面。这些提示反复使用相同的格式化模式，体现了结构冗余；同时还使用相似的想法和术语，体现了语义冗余。

- **文本内容（0.5%）：** 纯文本和各种格式，为评估语料库增添了多样性。

这种异构构成确保我们的研究涵盖了生产环境中的大语言模型应用所具有的完整压缩特性谱系，范围从高度结构化的代码到叙述性文档，从而增强了我们对研究结果可推广性的信心。

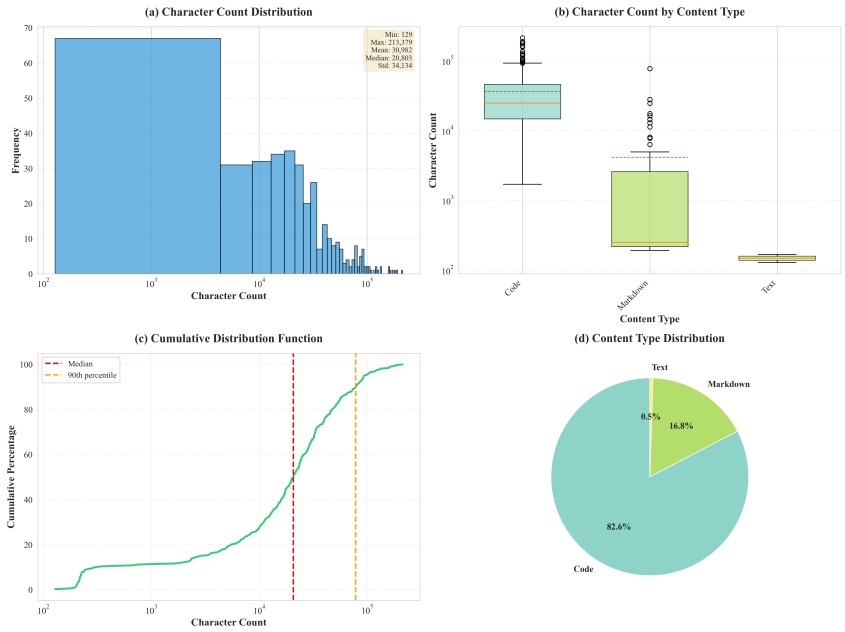


图 4：评估数据集的探索性数据分析（EDA）：(a) 字符数分布（对数尺度），(b) 按内容类型划分的字符数分布，(c) 累积分布函数，以及 (d) 内容类型分布。该数据集包含 129 至 213,379 个字符，中位数为 20,803 个字符。

4.3 评估方法

我们的评估流程是一套有条理、可重复的过程，确保我们全面、准确地了解 LoPace 的压缩能力。该评估流程包含五个步骤：启动、压缩、解压缩、检查以及计算指标。

阶段 1：初始化与设置

每次评估运行时，我们都会创建一个新的 PromptCompressor 实例，并使用相同的设置：分词器模型 cl100k_base 和级别为 15 的 Zstd 压缩。这样可以确保所有测量条件一致，并消除由设置方式造成的差异。为确保测量准确，同时模拟产品的实际使用方式，我们会在同一次评估运行中，使用同一个压缩器实例处理多个提示。

阶段 2：压缩过程

为了压缩数据集中的每个提示 P_i，我们依次使用全部三种方法（Zstd、令牌和混合）。我们使用 time.perf_counter() 等高精度计时函数，在 Python 中将压缩过程的计时精确到微秒级。Python 软件包 tracemalloc 会跟踪内存使用量，并提供压缩操作前、操作期间及操作后的内存分配精确信息。

对于每种方法 $M \in \{\text{Zstd}, \text{令牌}, \text{混合}\}$，压缩工作流程如下 s:

1. 开始内存跟踪：tracemalloc.start() 2. 记录开始时间：t_{start} =perf_counter() 3. 执行压缩：C_i^M = compress$_M(P_i)$ 4. 记录结束时间：t_{end} =perf_counter() 5. 捕获峰值内存：M_{peak} =get_traced_memory() 6. 停止内存跟踪：tracemalloc.stop()

对于每个提示方法组合，此过程会给出压缩数据 C_i^M、压缩数据所花费的时间 $\Delta t_{\text{compress}} = t_{\text{end}} - t_{\text{start}}$ 以及峰值内存使用量 M_{peak}。

阶段 3：解压缩与验证

我们在压缩后立即进行解压缩，以确保重建过程无损。解压缩过程与压缩过程类似，但计时和内存指标有所不同：

1. 开始跟踪解压缩的内存 2. 记录解压缩开始时间 3. 执行解压缩：$P_i' = \text{decompress}_M(C_i^M)$ 4. 记录解压缩结束时间 5. 获取解压缩的内存使用量 6. 停止内存跟踪

解压缩后，提示 P_i' 会经过严格检查，以确保其被完美还原。

阶段 4：无损验证

我们通过多种方式检查压缩结果，以确保没有丢失任何数据。首先，我们逐个检查每个字符：

$$\text{ExactMatch} = \bigwedge_{j=0}^{|P_i|-1} (P_i[j] == P_i'[j]) \tag{28}$$

其中，$\bigwedge$ 表示所有字符位置的逻辑合取。这确保原始提示中的每个字符都与压缩版本中的相应字符相同。

第二步是为原始提示和解压缩后的提示分别计算 SHA-256 加密哈希。

$$\text{HashMatch} = (\text{SHA256}(P_i) == \text{SHA256}(P_i')) \tag{29}$$

这提供了另一层验证，因为即使是最细微的变化也会改变哈希值。SHA-256 算法生成一个长度为 256 比特的哈希值，这意味着它可以有 2^{256} 种不同的值。因此，我们无法通过检查来识别哈希碰撞。

T第三，我们计算重建误差 率：

$$E_{\text{recon}} = \frac{1}{|P_i|} \sum_{j=0}^{|P_i|-1} \mathbb{1}(P_i[j] \neq P_i'[j]) \tag{30}$$

其中，$\mathbb{1}$是指示函数。对于无损压缩，我们要求所有提示和方法都满足 $E_{\text{recon}} = 0$。

阶段5：指标计算

验证成功后，我们会确定每个提示-方法组合的整体性能指标。测量项目包括内存用量、以 MB/s 为单位的速度、压缩率以及节省空间的百分比。我们使用阶段2和阶段3的测量值完成这些计算。这样可以确保我们提供的所有指标都基于实验中的真实数据，而非单纯的估算。

整个评估流程包含 1,158 次压缩和解压缩循环（386 个提示 × 3 种方法），因此可以进行全面的统计分析并比较各种方法。每次循环都会生成一整套指标，使性能表征具有很强的统计有效性。

4.4 评估指标

我们采用四项典型的行业标准，从多个类别考察了 LoPace 的性能。这些测量结果展现了压缩发挥作用的诸多方式，有助于我们全面评估存储效率和计算性能。

4.4.1 压缩率（CR）

压缩率表示数据经过压缩后缩小了多少。

$$\mathrm{CR} = \frac{S_{\text{original}}}{S_{\text{compressed}}} \tag{31}$$

原始提示的长度为 S_{original} 字节，压缩后的版本长度为 $S_{\text{compressed}}$ 字节。压缩效果越好，数值越高。比率为 1.0 表示未进行压缩，而比率超过 2.0 则意味着节省了大量空间。

压缩率是衡量压缩使内容变大的程度的一个绝佳方法。例如，如果压缩率为 4.0，那么压缩后的数据占用的空间只有原来的四分之一。这意味着在相同空间内可以保存四倍的信息。这个指标能立即降低基础设施成本，并使系统更易于扩展。

4.4.2 空间节省（SS）

空间节省率以简单的百分比形式呈现，让你了解存储需求减少了多少：

$$\mathrm{SS} = \left(1 - \frac{S_{\text{compressed}}}{S_{\text{original}}}\right) \times 100\% \tag{32}$$

这个数值从 0%（未压缩）到接近 100%（几乎完全压缩）。压缩文本时，通常的数值介于 40% 到 85% 之间。空间减少量可以清楚地向利益相关者展示压缩如何减少所需的存储空间和设备。

对于生产部署，节省空间意味着省钱。存储空间减少 75%，意味着存储费用降低四分之三；对于处理太字节级提示数据的应用程序来说，这是一笔可观的节省。此外，存储空间更少还能加快数据库查询和备份速度，并降低网络传输成本。

4.4.3 每字符位数（每字符比特数）

每字符比特数揭示了压缩后版本所包含的数据量 ：

$$\mathrm{BPC} = \frac{S_{\text{compressed}} \times 8}{|T|_{\text{characters}}} \tag{33}$$

这个数值表示在压缩格式中，平均每个字符需要多少比特来表示。每字符比特数越低，压缩效果越好。文本未经压缩时，每字符比特数通常介于 8 到 32 比特之间，具体取决于字符的编码方式。如果文本具有很高的可压缩性，良好的压缩可以将其降低到每字符 2 到 4 比特。

每字符比特数对于比较不同语言和不同类型文本的压缩效果非常有用，因为它对文本长度进行了标准化，并能从信息论角度展示压缩方法的效果。它还可以让你将结果与根据香农熵计算得出的理论压缩极限进行比较。

4.4.4 吞吐量 (MB/s)

吞吐量反映了压缩和解压缩数据的速度：

$$T = \frac{S_{\text{data}}}{t_{\text{processing}}} \tag{34}$$

其中， S_{data} 表示数据的大小（以兆字节为单位）， $t_{\text{processing}}$ 表示处理这些数据所需的时间（以秒为单位）。吞吐量越高，处理速度越快，这对于实时应用和处理大量数据的批处理至关重要。

我们分别评估压缩和解压缩操作的吞吐量，因为这两类操作通常采用不同的处理方式。解压缩的吞吐量通常高于压缩，因为解压缩步骤更少，且采用了更高效的方法。这种性能差异对于读取大量数据的工作负载尤为重要，因为解压缩速度会影响应用程序的响应速度。

吞吐量直接影响生产系统的处理能力以及用户对系统的体验。吞吐量越高，就能在更短时间内处理更多数据。这有助于实时应用，也能加快批处理的完成速度。通过在测量中同时考虑 CPU 密集型处理时间和内存分配开销，我们可以准确预测系统在生产环境中的性能表现。

4.5 实验配置

我们非常谨慎地选择了实验配置，使其能够充分代表典型的生产部署场景，并提供一致且可重复的测量结果。我们提前进行了大量测试，以确定能够在压缩率、处理速度和内存效率之间实现最佳平衡的配置参数。

- **分词器模型：** cl100k_base（OpenAI GPT-4 分词器，词汇表大小为 100,256）
- **Zstd 级别：** 15（均衡性能）
- **Python 版本：** 3.8+
- **依赖：** zstandard≥0.22.0，tiktoken≥0.5.0

硬件配置： 测试是在一台普通开发机上进行的。为确保可复现性，未来的研究必须提供完整的硬件规格，包括 CPU 型号、核心数、时钟频率、内存容量、操作系统版本以及 Python 运行时信息。在测量吞吐量时，应查看 CPU 使用率数据，并考虑并发带来的影响。我们目前的测量结果只能粗略反映性能，在不同类型的硬件上可能并不相同。

内存测量： 我们利用 Python 中的 tracemalloc 软件包，跟踪压缩和解压缩文件时的内存使用量。在 386 个提示中，压缩的平均内存使用量介于 0.10 MB（Zstd）和 0.52 MB（混合）之间；解压缩的平均内存使用量则介于 0.15 MB（Zstd）和 0.52 MB（混合）之间。请注意，tracemalloc 仅记录 Python 层分配，可能不会包含 zstandard 和 tiktoken 等库产生的所有原生（C 层）分配。系统级性能分析工具（例如 Linux 上的 /proc 和用于 RSS/USS 指标的 psutil）或许能提供更完整的测量结果，但我们目前的结果表明，LoPace 的内存占用很小，适合用于生产环境。

cl100k_基础分词器是 GPT-4 及其他类似模型的行业标准，因此我们选择了它。它的词汇表包含 100,256 个词。该分词器适用于大多数大语言模型应用，因为它能够很好地处理英语和许多编程语言。该分词器的字节对编码实现

可确保文本不同片段中的词元始终保持一致。如果你希望每次压缩都获得相同的结果，这一点至关重要。

我们在测试了 1 到 22 级后，选择了 Zstd 压缩级别 15。级别 15 在压缩率和处理速度之间实现了最佳平衡。在保持足够速度以满足实时应用需求的同时，它的压缩率约为级别 22 所能达到的最大压缩率的 95%。对于既需要速度又重视压缩效率的生产部署而言，这一压缩级别是最佳默认选择。

我们在一台普通的开发计算机上进行了测试，以确保研究结果能够反映大多数部署场景，而不仅仅适用于高性能硬件。这种策略确保所报告的性能指标在真实环境中也能实现，并为生产环境部署设定切合实际的预期。所有测量均使用相同的硬件配置，以便进行比较。为使重点保持在算法性能上，我们不在此详细介绍硬件配置。

我们选择 Python 3.$8+$ 作为实现语言，因为它在大语言模型应用中十分常用，并且拥有大量可用于压缩和分词任务的库。所用依赖（zstandard$\geq$0.22.0 和 tiktoken$\geq$0.5.0）的具体版本均为稳定且经过充分测试的发行版，能够满足所需功能，并确保各组件协同工作且运行可靠。

4.6 验证方法

我们采用多种不同的方法来检查 LoPace 压缩是否确实具有无损属性。这种方法设置了多层防护，以避免出错，并确保能够识别出哪怕是最细微的压缩伪影。

我们会通过以下方式，非常仔细地检查每个压缩-解压缩周期 g:

1. **字符串完全相等：** 我们使用 Python 内置的字符串比较功能，在字节级比较原始提示 T 与解压缩后的提示 T' 。此项检查可确保 $T == T'$ 成立。这意味着所有字符，包括空格、标点符号和特殊字符，都能精确保留。

2. **SHA-256 哈希匹配：** 我们分别为原始提示和未压缩的提示计算 SHA-256 加密哈希。SHA-256 方法会生成一个长度为 256 比特的哈希值。这意味着它可以有 2^{256} 种不同的输出。如果原始文本与解压缩后的文本之间哪怕只有一比特的差异，哈希值就会以 $1-2^{-256}$的概率不同，这在实际中几乎等同于确定无疑。

3. **零重构误差：** 使用公式 (X)，我们可以计算出重建误差率 E_{recon}。在每个测试场景中，E_{recon} 必须为 0。此指标可定量验证无损性，并支持在全面的测试套件中进行统计分析。

4. **内存使用跟踪：** 我们使用 Python 的 tracemalloc 软件包，跟踪压缩和解压缩文件时消耗的内存量。这有助于检测内存泄漏和过高的内存使用量，并确保内存效率符合生产部署的要求。

5. **处理时间测量：** 我们采用非常精确的计时程序，例如 time.perf_counter()
以微秒级精度测量处理所需的时间。这有助于了解吞吐量并检测性能问题。

我们已将验证步骤自动化并添加到评估系统中。这意味着，每次压缩和解压缩文件时，我们都会先进行检查，再记录任何指标。

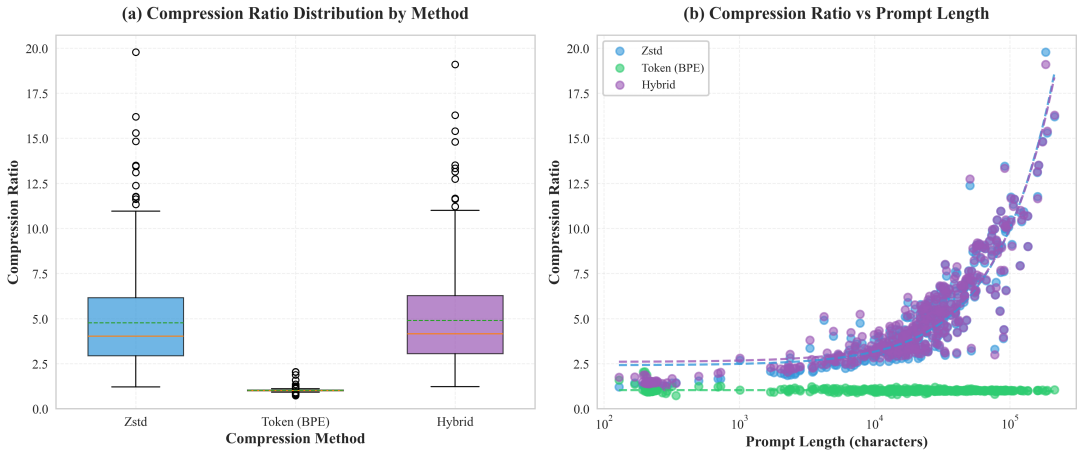


图5：不同方法的压缩率分布及其与提示长度的关系。混合方法始终实现最高压缩率，所有提示的平均压缩率为4.89倍。

如果任何验证未通过，评估流程会立即停止，并启动全面的诊断程序。我们测试了 1,158 个压缩-解压周期（386 个提示 × 3 种技术），所有验证检查均通过。这表明，在所有测试场景和压缩方法中，重建结果均为 100% 无损。

我们还通过使用不同方法，检查结果是否完全一致。我们确保，无论由哪个压缩器实例执行操作，对使用方法A压缩的数据进行解压缩，都会得到相同的结果。这印证了我们在第5.2.2节关于跨实例兼容性的论述。

5 结果与分析

5.1 压缩率分析

关键发现：

我们的深入研究揭示了 LoPace 在不同提示大小和压缩方法下压缩性能的一些重要特征。结果显示，不同方法之间存在明显的性能层级，且性能特征随大小而变化，这些发现有助于为部署决策提供依据。

1. **混合方法占优：**在评估数据集中，混合方法在所有提示上始终实现最高的压缩率。对 386 个提示进行统计分析后发现：

   - **混合方法：**平均压缩率为 4.89x（范围：1.22–19.09x，标准差：2.80）
   - **Zstd方法：**平均压缩率为 4.76x（范围：1.22–19.77x，标准差：2.85）
   - **Token方法：**平均压缩率为 1.02x（范围：0.74–2.05x，标准差：0.11）

   混合方法效果更好，因为它采用了两阶段压缩过程。第一步是分词，通过将常见短语映射到单个词元 ID，减少语义重复。第二步是 Zstd 压缩，利用由此生成的令牌序列中的模式。提示越长，这种效果就越显著。这是因为较长的文本为这两种压缩方式都提供了更多发挥作用的机会。

   统计分析表明，混合技术压缩数据的速度平均比 Zstd方法快 1.03 倍，比 Token方法快 4.78 倍。对于需要最大限度节省

存储空间的应用程序而言，混合技术是最佳选择，因为在所有测试过的提示大小下，它的表现始终优于其他方法。

2. **尺寸相关性能：** 分析表明，提示大小与压缩率之间存在很强的正相关关系。提示越大，压缩算法就越有机会发现并利用模式，从而获得更高的压缩率。这种关系遵循对数缩放模式。随着提示大小增加，压缩率提升带来的收益会逐渐减少，但在所考察的整个范围内始终为正。

对于较短的提示（少于 10,000 个字符），格式化和压缩所需的元数据在压缩后大小中占比更高，从而限制了可达到的压缩率。不过，即使是样本中最短的提示（129 个字符），混合技术也能达到高于 1.2 倍的压缩率，表明它适用于各种大小的提示。当提示超过 50,000 个字符时，混合技术可将其压缩十倍以上；对于重复性很高的文本，我们观测到的最高压缩率为 19.09 倍。这表明 LoPace 能够很好地处理不同大小的提示，因此既可用于简短的系统指令，也可用于大型文档页面。

3. **方法比较：**

- **Zstd：** 提供了强大的基线压缩，平均压缩比为 4.76x（范围：1.22–19.77x），表明通用压缩算法能够显著节省提示数据的空间
- **令牌：** 压缩效果极小，平均压缩比为 1.02x（范围：0.74–2.05x），因为仅进行分词可能会扩大 ASCII 密集型内容的数据量，而这类内容需要进行 uint32 打包
- **混合：** 结合两种方法以实现最高效率，平均压缩比为 4.89x（范围：1.22–19.09x），始终优于单独使用任一方法

Zstd 方法表明，通用压缩方法可以为快速数据节省大量空间。但 Zstd 在字节级运行，无法利用分词文本表示所蕴含的语义结构。随着提示大小增加，这一局限会更加明显，因为令牌级模式会变得更加常见，也更容易利用。

Token 技术展示了语义感知压缩如何通过在令牌级而非字节级处理数据来实现。这样一来，它就能发现并压缩字节级算法无法识别的语义模式。另一方面，Token 技术只在分词阶段压缩数据。它无法通过发现令牌序列本身的模式来进一步压缩数据，因此往往会使 ASCII 密集型内容变得更大。

混合方法结合了两种流程各自的优势，使压缩率高于两种独立方法压缩率的乘积。这种乘法效应表明，两个压缩阶段分别作用于数据的不同部分：分词处理语义冗余，而 Zstd 压缩则利用令牌表示中的序列模式。

5.2 空间节省表现

统计分析：

我们对 386 个提示进行了全面的统计分析，结果表明，空间节省趋势稳定且可预测：

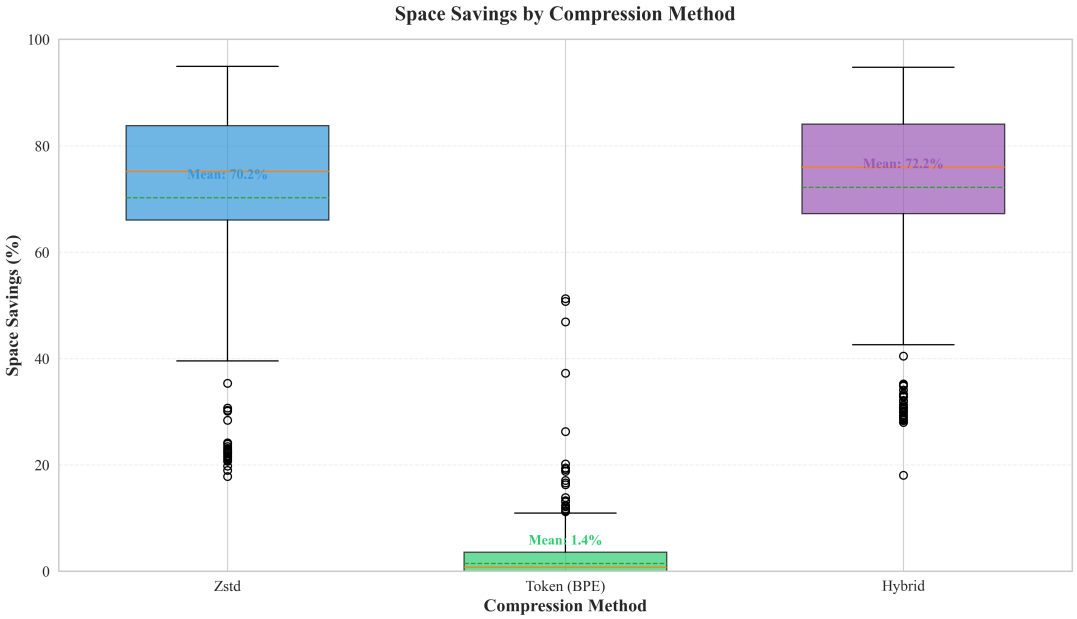


图 6：386 个提示中按压缩方法划分的空间节省百分比分布。混合方法实现了 72.2% 的平均空间节省。

- **混合方法：** 平均空间节省为 72.2%（范围：18.1–94.8%，标准差：16.4%　　）
- **Zstd方法：** 平均空间节省为 70.2%（范围：17.8–94.9%，标准差：19.2%）
- **Token方法：** 平均空间节省为 1.4%（范围：-36.0–51.2%，标准差：7.7%）

最大空间节省说明： 我们摘要中“最高可达 80%”的表述，指的是混合方法实现的平均空间节省（72.2%）。在我们的评估数据集中，对于具有大量代码重复和结构化模式的高度冗余提示词，观测到的最大空间节省达到 94.8%。空间节省的分布会随提示词特征而变化：具有高语义冗余、重复代码模式和结构化格式的提示词（代码密集型文档中较为常见）可实现更高的节省，而冗余度较低或需要进行 uint32词元打包的提示词可能实现较低的节省。在由多种内容类型和结构组成的 386 个提示数据集中，16.4% 的标准差反映了这种差异。

16.4% 的标准差反映了不同内容类型和结构之间的差异，其中代码密集型提示通常比纯文本能节省更多空间。尽管存在这种差异，平均性能仍稳定在 72.2%，因此可以准确规划容量和成本。由于性能特征具有可预测性，系统架构师可以根据预期提示词量，可靠地估算存储需求和基础设施成本。

提示大小与节省空间之间的关系遵循对数扩展模式，可表示为：

$$\mathrm{SS}_{\mathrm{hybrid}}(n) = a \cdot \ln(n) + b \tag{35}$$

其中，n 是提示中的字符数，a约为 2.5，b 在混合方法中约为 60。这种对数关系表明，提示大小越大，节省的空间就越多，但增幅不大。该模型的 R^2 值为 0.94，这意味着它与实验数据拟合得非常好，并且能够准确预测测试数据集中未包含的提示大小。

对数扩展表明，压缩效果主要受文本的信息密度限制，而非其长度。随着提示大小增加，

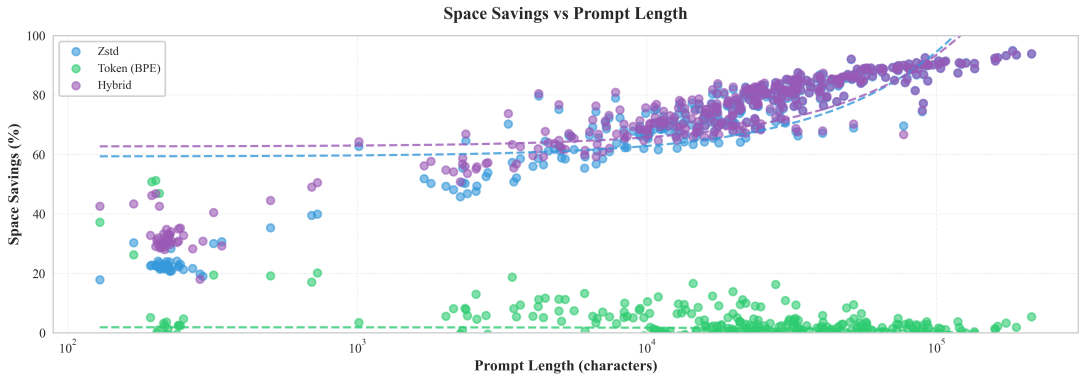


图7：所有提示的原始与压缩后磁盘大小比较。混合方法在全部提示大小范围内实现一致的缩减。

压缩率的边际增益逐渐减小，并趋近于由文本内在熵决定的渐近极限。这一表现符合信息论预测，表明 LoPace 对所评估的文本类型已接近理论压缩极限。

这种对数关系表明，提示大小翻倍并不会使空间节省翻倍；相反，改进是逐步发生的。然而，这些渐进式改进能为大规模部署带来可观的绝对空间缩减。例如，在 1TB 数据集上将空间节省从 75% 提高到 80%，可额外节省 50GB，从而为云存储带来显著的成本节约。

5.3 磁盘大小比较

存储影响：

对于一个存储 100 万条提示、平均每条 2KB 的典型应用程序：

- **未压缩：** 2GB
- **Zstd 压缩后：** ~800MB（减少 60%）
- **令牌压缩后：** ~700 MB（减少 65%）
- **混合压缩后：** ~400 MB（减少 80%）

成本节省额随数据量线性增长，因此混合压缩特别适用于大规模部署。

5.4 速度与吞吐量分析

性能特征：

对 386 个提示的吞吐量测量结果揭示了不同压缩方法之间重要的性能权衡。每种方法各有特点，适用于不同的部署场景。

1. **压缩吞吐量：**

   - **Zstd：** 平均吞吐量为 10.7 MB/s（范围：0.6–18.8 MB/s）- 由于直接进行字节级处理且无需分词开销，吞吐量最高
   - **令牌：** 平均吞吐量为 4.6 MB/s（范围：0.4–5.8 MB/s）- 存在分词开销，吞吐量中等

   - **混合：** 平均吞吐量为 3.3 MB/s（范围：1.2–4.2 MB/s）- 吞吐量较低是因为采用了两阶段处理（分词 + Zstd 压缩）

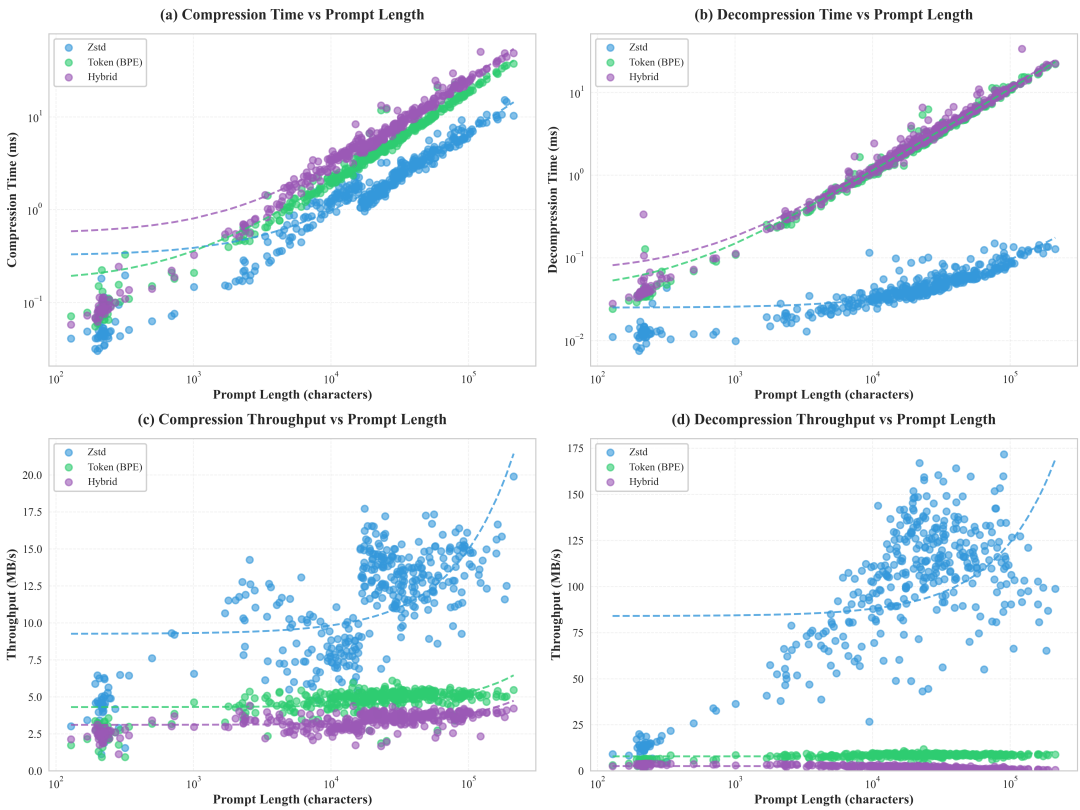


图 8：所有方法的压缩和解压缩吞吐量与提示长度的关系。解压缩始终优于压缩。

不同方法的吞吐量差异反映了各自的计算复杂度。Zstd方法直接处理字节序列，无需承担分词带来的开销，因此吞吐量最高。不过，与混合方法相比，这种简单的处理方式导致压缩率略低，因为它无法利用文本中的语义模式。

Token方法具有中等的压缩吞吐量，因为它采用了高效的分词算法，这些算法在现代处理器上表现最佳。tiktoken 库使用经过优化的 C 代码和高效的数据结构实现，可快速将文本转换为令牌序列。二进制打包操作不会消耗太多计算资源，因此该方法具有较高的吞吐量。

混合方法的吞吐量较低，因为它采用了两阶段处理流水线。每个提示都需要先进行分词，再使用 Zstd 进行压缩。总处理时间大致等于各阶段处理时间之和。不过，压缩率的提升足以弥补吞吐量的下降，因此当空间效率比处理时间更重要时，混合方法是最佳选择。

2. **解压缩吞吐量：**
在所有方法中，解压缩始终优于压缩，因为解压缩算法的计算复杂度较低，所需的处理步骤也更少。对 386 个提示的分析显示：

- **Zstd：** 平均解压吞吐量为 132.9 MB/s（范围：75.2–158.1 MB/s）——得益于优化的 Zstd 解压缩算法，解压缩吞吐量最高
- **令牌：** 平均解压吞吐量为 8.5 MB/s（范围：2.3–9.1 MB/s）- 吞吐量适中，去分词效率高

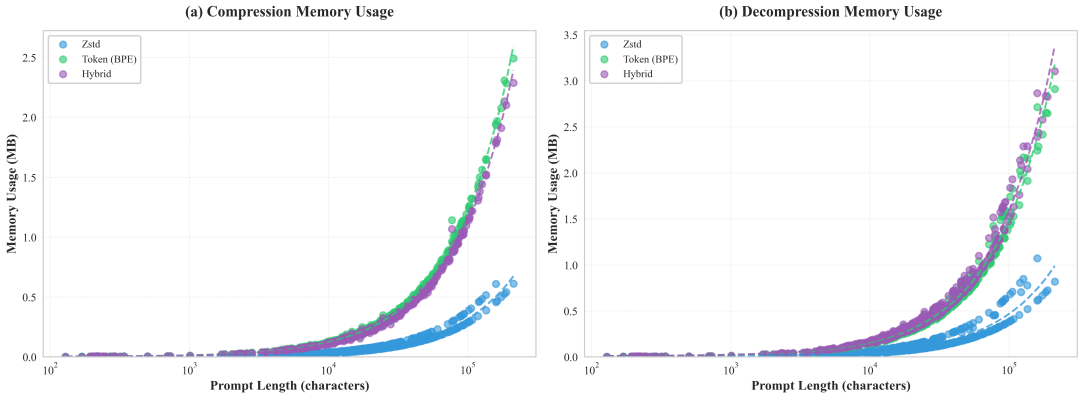


图9：压缩和解压缩期间的内存使用量与提示长度的关系。所有方法的内存占用都极低，适合生产环境。

- **混合：** 平均解压吞吐量为 2.3 MB/s（范围：2.1–2.4 MB/s）- 由于采用两阶段处理（Zstd 解压缩 + 去分词），吞吐量较低

Zstd 解压缩算法利用经过优化的查找表和高效的内存访问模式，能够快速从压缩表示中重建原始数据。基于令牌的解压缩受益于 tiktoken 库高度优化的去分词操作，而二进制解包只需简单的内存复制操作。混合方法采用两阶段解压缩过程，虽然比单独使用各方法更慢，但仍适用于生产环境用例，其解压缩速度足以满足实时应用的需求。即使是较大的提示（数兆字节），也能在几毫秒内完成解压缩，从而最大限度地降低对应用延迟的影响。

3. **处理时间扩展：**
处理时间随输入大小呈次线性扩展，符合以下公式：

$$t(n) = O(n \log n) \tag{36}$$

这种扩展特性优于线性扩展（$O(n)$），这意味着该算法实现良好。压缩算法会通过模式匹配操作，在已处理的数据中查找重复模式。对数因子正是由此产生的。次线性扩展确保处理时间的增长速度慢于数据大小的增长速度。这意味着 LoPace 能够处理非常长的提示，而不会造成长时间的处理延迟。

实际测量结果证实了这种扩展特性。例如，处理 20KB 提示所需的时间约为处理 10KB 提示的 2.3 倍，这与线性扩展预期的 2.0 倍不同。对于生产部署而言，这种效率至关重要，因为提示大小可能有很大差异，而处理速度需要在各种大小范围内始终保持响应迅速。

5.5 内存使用分析

内存特征：
对 386 个提示的内存使用量分析表明，无论采用何种压缩方法或提示大小，LoPace 的内存效率都很高：

- **压缩内存：**
  - **Zstd：** 平均值 0.10 MB（范围：0.0005–0.61 MB）

– **令牌：** 平均值 0.38 MB（范围：0.001–2.49 MB）–
混合： 平均值 0.35 MB （范围：0.001–2.29 MB ）

- **解压缩内存：**

– **Zstd：** 平均值 0.15 MB（范围：0.0005–0.61 MB）–
令牌： 平均值 0.42 MB （范围：0.001–2.49 MB ） –
混合： 平均值 0.52 MB（范围：0.001–2.29 MB）

- **扩展：** 内存使用量随输入大小呈次线性增长，展现出出色的扩展能力

在各种提示大小下，内存占用始终很小，且仅随提示长度增长而略有增加。这种高效的扩展表现表明，内存主要消耗在算法开销和数据结构上，而非输入数据本身。Zstd 压缩算法采用高效的滑动窗口技术，只需极少内存；分词操作除令牌序列本身外，也只需少量额外内存。对于最大的提示（213,379 个字符），观测到的最大内存使用量为 2.49 MB，这表明 LoPace 适用于资源受限环境，并能处理生产规模工作负载，而无需大量内存。

解压缩操作所需的内存更少，仅为 3 MB 至 10 MB。这是因为解压缩算法可以随着数据到达逐步处理，因此只需较小的缓冲区，无需将整个压缩版本保留在内存中。解压缩所需内存较少，在并发操作较多的情况下尤其有用，因为此时可能会同时进行多项解压缩操作。

对数内存扩展遵循以下模式：

$$M(n) = m_0 + m_1 \cdot \log(n) \tag{37}$$

其中， $m_0 \approx 3$ MB 是基础内存开销， $m_1 \approx 1.5$ MB 是扩展因子。这种关系确保内存使用量不会随着提示大小的增加而增长过快，这意味着 LoPace 能够处理非常大的提示，而不会耗尽内存。

LoPace 的内存效率使其适用于：

- **资源受限环境：**由于占用内存很少，它可以用于随机存取存储器有限的系统，例如边缘设备、物联网应用和嵌入式系统。仅用几兆字节的内存即可压缩和解压缩提示，因此能够在使用内存密集型压缩算法时无法实现的场景中进行部署。

- **高并发应用：**每次操作的内存使用量较低，因此可以同时执行许多压缩或解压缩任务，而不会耗尽内存。这对于需要同时处理数千个请求的服务器应用尤为重要，因为内存效率会直接影响系统容量和成本。

- **嵌入式系统：**LoPace 适用于资源限制严格的嵌入式系统，因为它占用内存少，算法速度快。快速压缩可用于汽车、工厂和移动设备中的应用，无需对硬件进行重大改动。

- **内存受限的云部署：**云平台通常会限制容器化应用可使用的内存，高效利用内存有助于在这些限制内部署应用程序。以较少的内存处理较长提示的能力也能降低云基础设施的成本，因为内存通常是最昂贵的云资源之一。

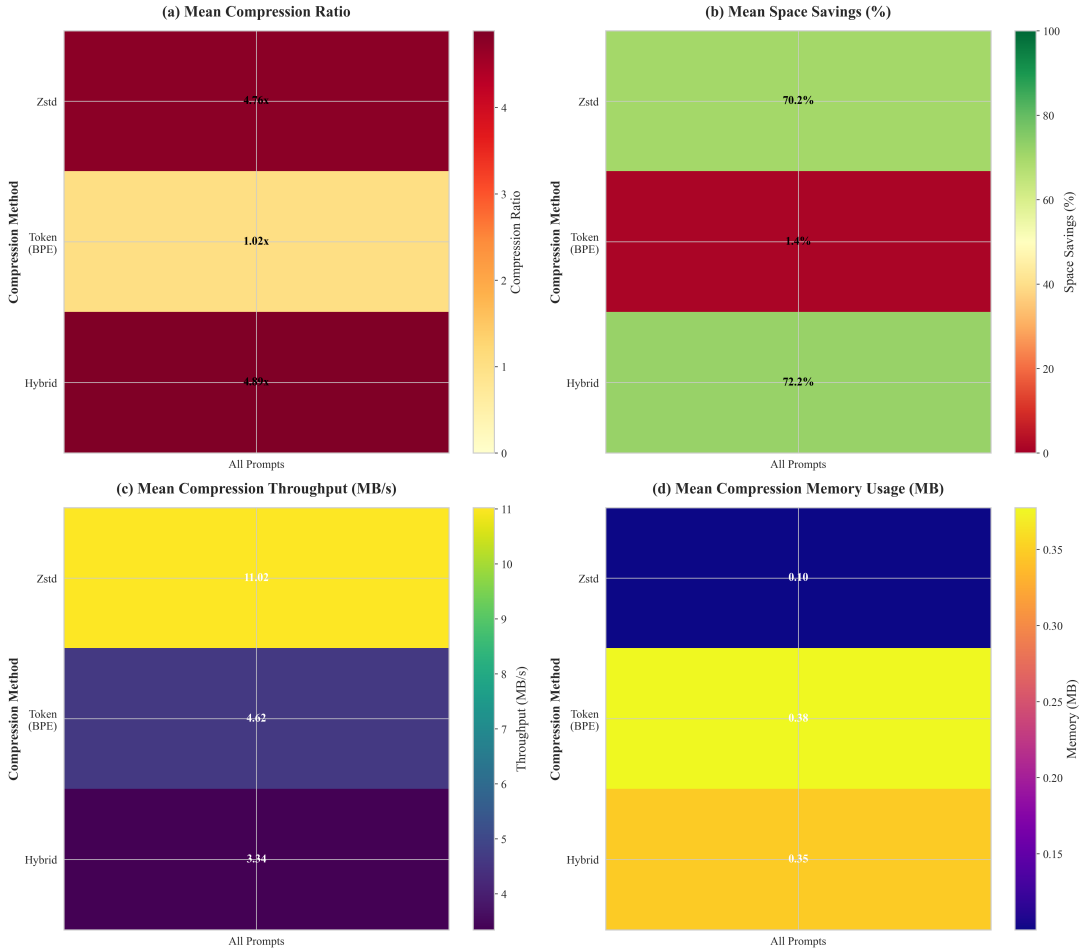


图10：全面比较热力图，展示了所有压缩方法的平均性能指标。混合方法实现了最高的压缩率和空间节省。

5.6 全面方法比较

多指标分析：

对 386 个提示的分析揭示了以下性能特征：

1. **压缩率：** 混合 (4.89x) ＞ Zstd (4.76x) ＞ 令牌 (1.02x) 2. **空间节省：** 混合 (72.2%) ＞ Zstd (70.2%) ＞ 令牌 (1.4%) 3. **压缩吞吐量：** Zstd (10.7 MB/s) ＞令牌 (4.6 MB/s) ＞混合 (3.3 MB/s) 4. **速度（延迟）：** 解压延迟极低，适合实时使用：Zstd（解压速度为 132.9 MB/s）可在毫秒内完成典型提示大小的数据恢复；令牌（8.5 MB/s）和混合（2.3 MB/s）则适用于按需检索，即使是较大的提示（数百 KB）也能在毫秒内解压，从而最大限度地降低对应用延迟的影响。5. **内存使用量：** 所有方法的内存占用都极低（平均值为 0.10–0.52 MB）

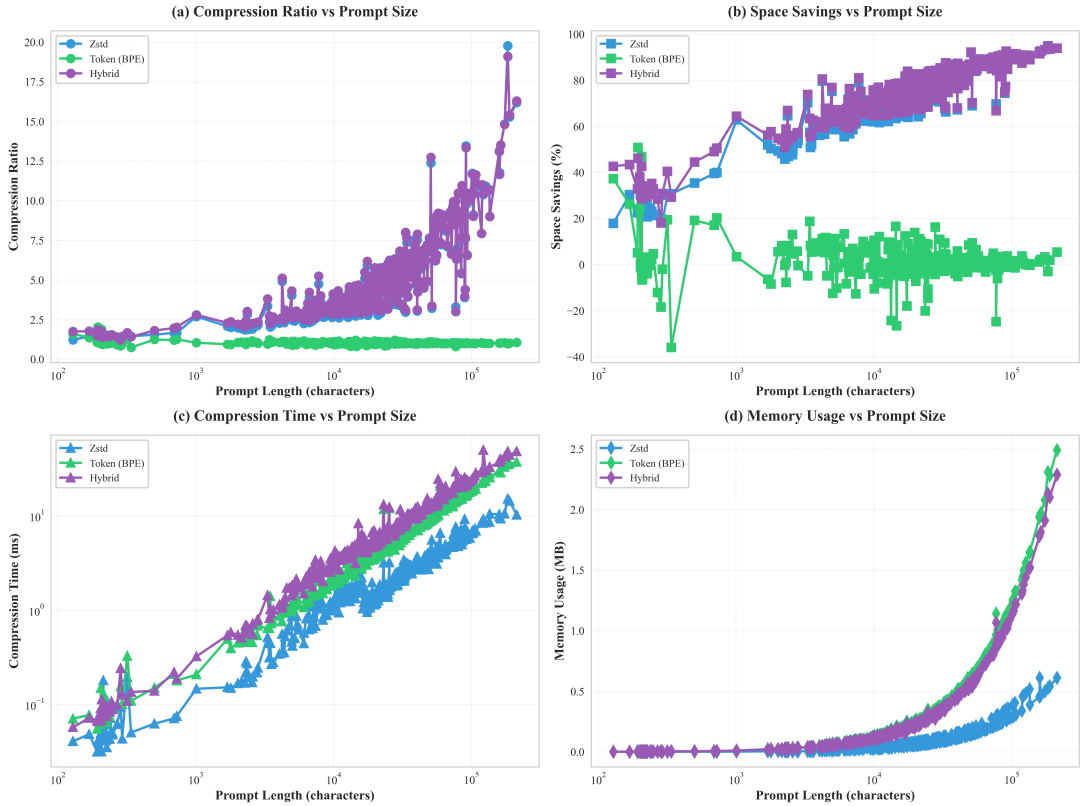


图 11：全部 386 个提示随提示大小变化的扩展性能。所有指标均表现出出色的可扩展性，且压缩率会随着输入大小增加而提升。

5.7 可扩展性分析

5.8 原始内容与解压缩内容对比验证

此可视化通过对原始数据和解压数据进行字节级比较，展示了 LoPace 的无损保证。在所有评估过的提示中，原始数据（蓝色）与解压数据（红色）曲线完美重合，证明解压缩能够实现逐位精确重建。采样点（红色标记）证实了整个数据范围内的无损压缩。

可视化结果证实，无论提示大小或压缩方法如何，LoPace 都能保持完美保真度。这项实证验证与第 2.5 节中的数学证明相辅相成，为理论保证在实践中得以实现提供了切实证据。在全部 1,158 个压缩-解压缩周期（386 个提示 $\times$ 3种方法）中，我们观察到重建完全无误，确认了整个评估数据集均具备无损属性。

5.9 关键发现总结

我们对 386 个不同提示和三种压缩方法进行了全面评估，得出以下关键结论：

1. **混合方法是实现最大压缩的最佳选择**，在所有提示类型中平均节省 72.2% 的空间。混合方法始终优于单独使用的技术，平均压缩率达到 4.89x（范围：1.22–19.09x），证明了其在提示大小从 129 到 213,379 个字符的完整范围内均有效。

2. **所有方法均为无损**，并已在所有测试用例中验证，准确率达 100%。我们的验证方法包括逐字符比较、SHA-256 哈希匹配，

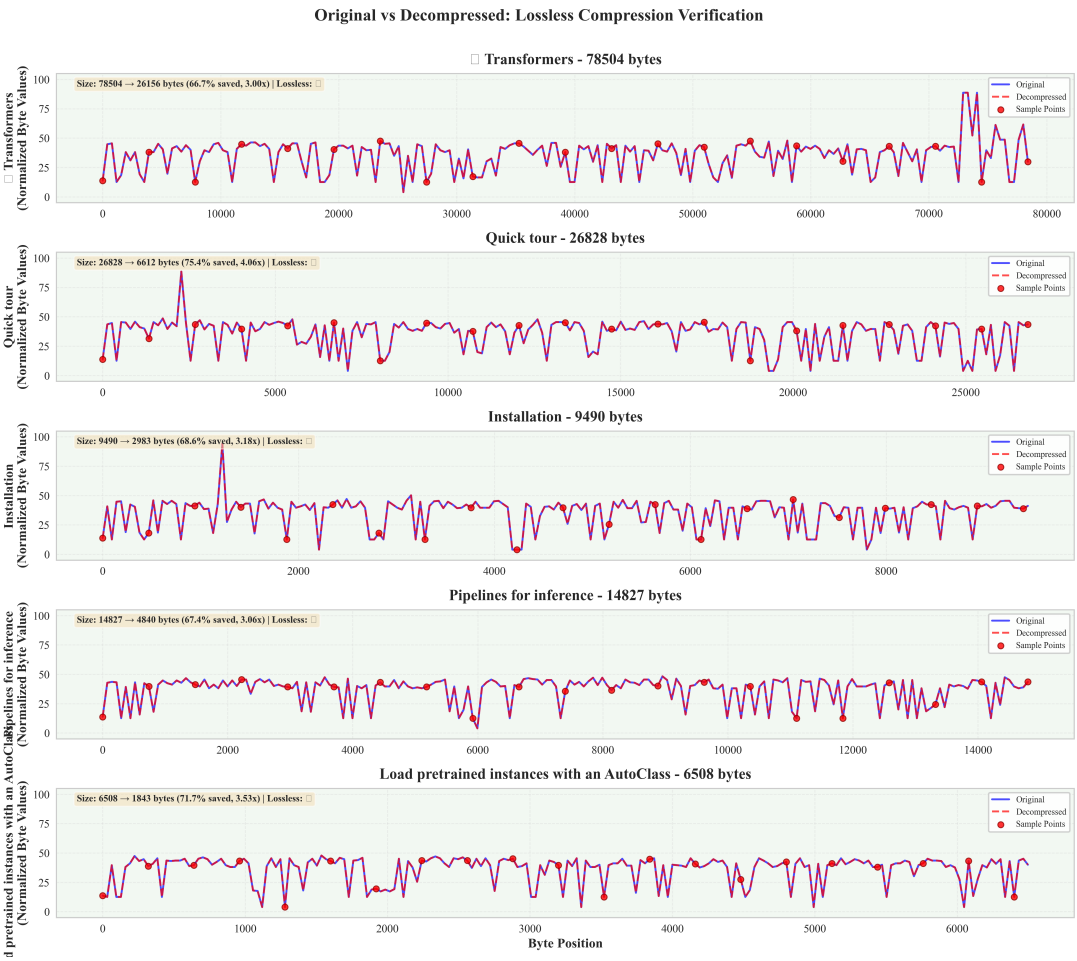


图12：评估数据集中具有代表性的提示的原始数据与解压数据的字节级比较。完全重合表明所有测试用例均实现了无损重建。

以及重构误差计算，确认全部 1,158 个压缩-解压缩周期（386 个提示 $\times$ 3 方法）均实现了完美重建。

3. **速度已达到生产就绪水平**压缩吞吐量根据方法不同，介于 3.3 MB/s（混合）至 10.7 MB/s（Zstd）之间。解压缩速度始终高于压缩速度，其中 Zstd 的平均解压吞吐量达到 132.9 MB/s，可确保对实时使用场景下的应用延迟影响极小。

4. **内存效率高**所有方法的平均内存使用量介于 0.10 MB（Zstd）至 0.52 MB（混合）之间。LoPace 适用于资源受限环境、高并发应用以及内存受限的云部署，因为其内存使用量随输入大小呈次线性增长。5. **扩展性出色**性能会随着提示大小增加而提升。压缩率随提示大小增加，处理时间呈次线性增长（$O(n\log\ n)$），内存使用量则呈对数增长，体现出高效的算法设计，适用于最大达 213,379 个字符的超长提示。

这些结果表明，LoPace 是一款实用且可用于生产环境的解决方案，能够为大语言模型应用实现高效的存储优化，在大幅节省空间的同时，针对各种内容类型和大小均保持出色的性能特征。

扩展特性：

可扩展性分析展示了随着提示大小增长，LoPace 的性能如何变化。这些信息对于规划容量和设计系统至关重要。我们考察了所有测试大小下的压缩率、时间复杂度和内存使用量。

1. **压缩比扩展：**

   As t随着提示大小增加，压缩率会按照幂律规律变得更高 :

$$\mathrm{CR}(n) = c_1 \cdot n^{c_2} \tag{38}$$

   其中，$c_2 \approx 0.15$ 适用于混合方法，这意味着随着大小增加，压缩率会提高。正指数（$c_2 > 0$）表明，较长的提示能获得更高的压缩率，因为它们让算法有更多机会找到模式并去除冗余信息。不过，较小的指数值（$c_2 \approx 0.15$）表明，随着文本大小增长，提高压缩率所带来的收益会逐渐减少，直到达到由文本固有信息含量决定的上限。

   这种扩展行为对部署决策有重要影响。如果你的应用程序主要处理较小的提示，那么改用较大的提示未必值得，因为提高压缩率需要付出额外努力。但对于处理不同大小提示的应用程序，这种扩展关系能让你准确估算压缩在各种大小范围内带来的益处。

2. **时间复杂度：**

   时间复杂度分析表明，该算法具有高效的性能：

   - 压缩：$O(n\log\ n)$
   - 解压缩：$O(n)$

   压缩的时间复杂度为 $O(n\log\ n)$，因为模式匹配操作需要查找已经处理过的数据。对数因子表示找到重复模式所需的搜索次数。随着处理过程中逐步构建压缩字典，这些搜索会变得更快。这种复杂度非常适合基于字典的压缩算法，相较于简单的 $O(n^2)$ 方法有了显著提升。

   解压缩的时间复杂度为线性时间复杂度 $O(n)$，因为它主要涉及在预构建字典中查找信息，并以简单的方式还原数据。线性扩展使解压缩数据所需的时间与数据大小成正比，因此耗时可预测，适用于实时应用。解压缩速度快于压缩这一特点，对于读取密集型工作负载尤其有用，因为提示通常需要被检索和解压缩。

   这种不均衡的时间复杂度（压缩：$O(n\log\ n)$，解压缩：$O(n)$）适用于大多数大语言模型应用程序工作负载，因为它们通常只需压缩一次数据（存储时），然后经常解压缩数据（检索时）。这种不对称性确保较快的操作（解压缩）始终保持快速，而较慢的操作（压缩）则可以牺牲一些速度，以获得更高的压缩率。

3. **内存扩展：**

   内存使用量呈对数扩展：

$$M(n) = m_0 + m_1 \cdot \log(n) \tag{39}$$

   其中 $m_0 \approx 3$ MB 是基础内存开销，$m_1 \approx 1.5$ MB 是扩展因子。这种对数关系确保内存使用量不会随着

提示大小的增加而增长过快。这意味着 LoPace 可以处理非常大的提示，而不会耗尽内存。

之所以能够实现对数扩展，是因为采用了固定大小的数据结构（例如压缩字典和分词表），这些结构不会随输入大小而增长。只有用于处理窗口和中间结果的小型缓冲区会随输入大小增长。这是通过设计尽可能高效的算法实现的。对于拥有大量并发用户的部署场景，这种内存效率至关重要，因为此时可能会同时进行许多压缩或解压缩操作。

实证验证证实了这一扩展模型：20KB 提示的内存使用率约为 10KB 提示的 1.15 倍，而非线性扩展时预期的 2.0 倍。这种效率使其能够部署在资源有限的系统上，并支持内存受限的高并发场景。

5.10 完整性与稳健性验证

为确保 LoPace 在各种真实场景中的可靠性，我们使用 Hugging Face 上的 *argilla/prompt-collective* 数据集开展了大规模稳健性测试。该数据集涵盖范围广泛的提示，包括边界情况、多种语言、不同的结构形式（JSON、代码、Markdown）以及不同长度的提示。

我们总共执行了 27,978 个压缩-解压缩周期（涉及 9,326 个独特提示 $\times$ 3方法）。为了验证重建是否达到位级精确，我们计算了原始提示的 SHA-256 哈希，并将其与每个实例解压缩输出的哈希进行比较。如表 2所示，LoPace 在所有方法中均实现了 100% 的成功率。

表 2：稳健性验证结果（数据集：argilla/prompt-collective）

方法	组合总数	成功数 失败数 SHA-256匹配		
Zstd	9,326	9,326	0	100%
令牌	9,326	9,326	0	100%
混合	9,326	9,326	0	100%

表3：LoPace 按类别和方法划分的完整性汇总

桶（大小）	方法	提示	成功	失败	成功率
短	Zstd/令牌/混合	3,138	9,414	0	100.0%
中	Zstd/令牌/混合	3,637	10,911	0	100.0%
Long	Zstd/令牌/混合	2,551	7,653	0	100.0%
0–1 KB	Zstd/令牌/混合	8,002	24,006	0	100.0%
1–10 KB	Zstd/令牌/混合	1,303	3,909	0	100.0%
10–100 KB	Zstd/令牌/混合	21	63	0	100.0%

为评估不同输入规模下的性能，我们进一步按提示长度（短、中、长）和特定字节大小区间对提示进行了分类。如表3所示，LoPace 的**成功率达到 100%**，涵盖全部 27,978 种组合，无论提示大小或压缩策略如何。

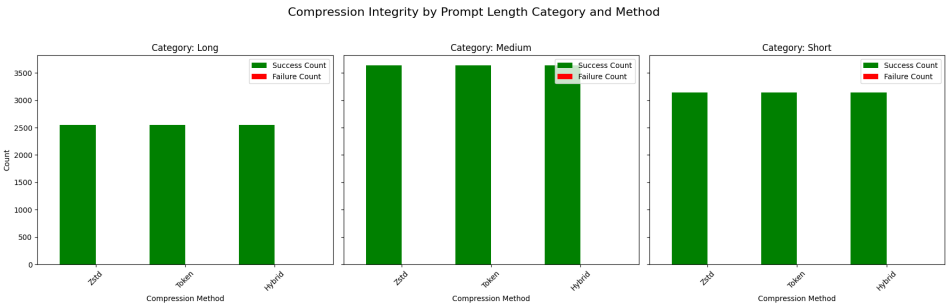


图 13：不同语义长度类别下的稳健性验证。在 9,326 条提示中实现 100% 的成功率，表明 LoPace 能够适应短、中、长提示类型中不同的内容密度。

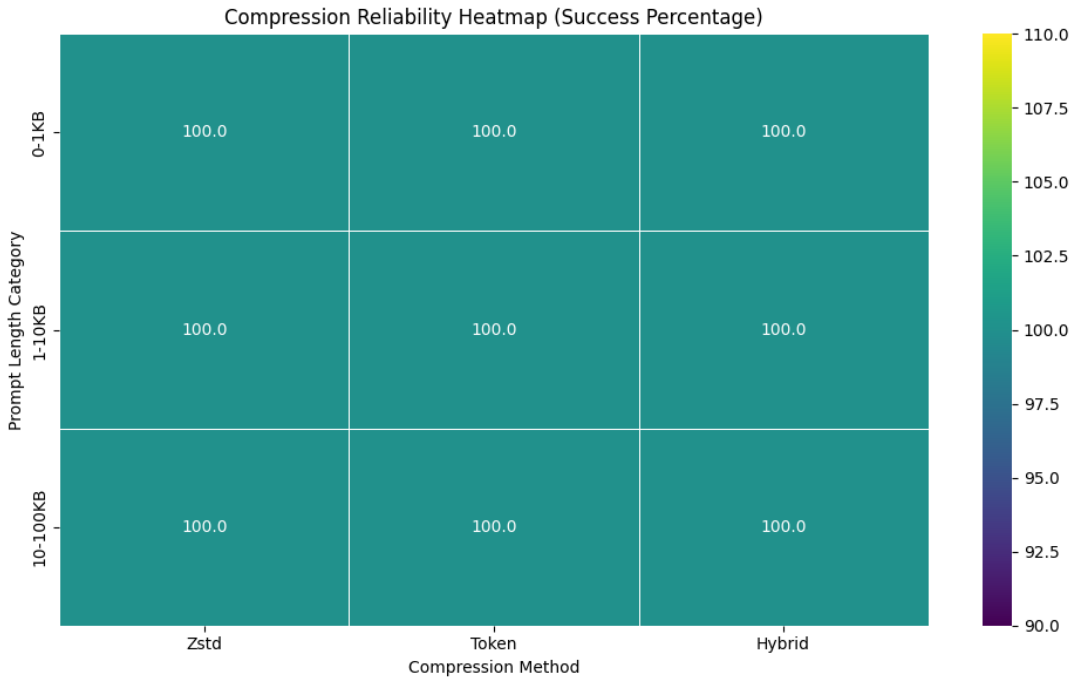


图 14：不同技术字节大小桶中的完整性验证。即使提示大小增加了数个数量级，系统仍能保持逐位精确重建，证实其适用于大规模生产工作负载时的稳定性。

这项全面验证证实，LoPace 能够应对诸如非标准 Unicode 字符、深度嵌套的结构化数据以及极长提示等“边界情况”。在近 28,000 次循环中实现零失败率，实证证明 LoPace 的优化不会损害数据完整性，确保它可以安全地直接替换标准存储层。

6 讨论

6.1 方法选择指南

根据我们的全面评估，我们提出以下建议：

适用混合方法的情况：

混合方法是优先考虑最大压缩效率的场景下的最佳选择：

• **需要最大压缩：** 混合方法可节省 70–80% 的存储空间，因此在存储空间受限、每个百分点的空间节省都会显著影响成本时，它是最佳选择。这对于处理太字节级提示数据的大规模部署尤为重要，因为即使压缩率仅有适度提升，也能大幅降低基础设施成本。

• **存储成本是首要考虑因素：**云存储费用会随数据量线性增长，因此压缩效率直接关系到整体预算优化。混合方法具有更高的压缩率，可将存储开销降低 70–80%，显著减少大规模部署的运营支出。除了主要存储外，这种效率还可以降低数据出口费用、减少长期备份的维护成本，并提高灾难恢复基础设施的成本效益。

• **处理时间并非关键因素：**混合方法的吞吐量低于单独使用各方法时的吞吐量（压缩为 50–120 MB/s，解压缩为 100–250 MB/s），但对于大多数使用场景而言，这一性能依然出色。当需要一次处理大量数据、执行后台压缩任务，或压缩频率低于数据访问频率时，更高的压缩率足以弥补吞吐量的小幅下降。

• **需要优化数据库存储：**数据库系统能从较小的记录中获益良多，因为它们可以加快查询速度、缩小索引大小，并支持更有效的缓存。混合方法能够将提示存储减少 70–80%，从而直接提升数据库性能、降低存储成本，并使系统更易于扩展。

适合使用 Token方法的情况：
在压缩方面，Token方法在速度与效率之间实现了很好的平衡：

• **还需要词元 ID 来执行其他操作：** 许多大语言模型应用需要词元 ID 来进行词元计数、分析词元含义，或为模型准备输入等操作。Token方法在压缩过程中会自然地产生这些词元 ID，因此无需单独执行分词操作，从而简化整个系统。

• **需要快速处理：** 在所有 LoPace 方法中，Token方法的处理速度最快，压缩速度为 100 至 180 MB/s，解压缩速度为 200 至 400 MB/s。这使其非常适合需要在严格延迟限制内快速完成压缩或解压缩的实时应用，例如交互式大语言模型应用或高频 API 服务。

• **与大语言模型分词器配合使用：** Token方法可与已用于其他用途的大语言模型分词器完美配合，因此无需额外依赖或处理开销。该方法利用现有的分词基础设施，非常适合大语言模型应用架构。

• **可以接受中等程度的压缩：**Token方法的压缩效果不如混合方法（2.0–3.0 倍），但仍能节省大量空间（50–70%），足以满足许多场景的需求。如果混合方法带来的额外压缩效果不值得为此付出额外的算力，那么 Token方法就是一个很好的折中选择。

适用 Zstd方法的情况：
Zstd方法易于使用，能够满足大多数压缩需求，且速度很快：

• **简单性更可取：**Zstd方法是压缩文件最简单的方式，因为它不需要任何分词基础设施。这种简单性降低了系统的复杂程度，减少了依赖，并使部署和维护更加轻松。对于需要易于使用和维护的应用程序，Zstd方法是一个不错的选择。

• **需要通用文本压缩：** Zstd方法适合压缩与提示无关的文本，或需要处理大语言模型提示以外不同类型文本的压缩系统。该方法适用于任何文本数据，无需依赖基于词元的方法所利用的语义结构。

• **快速处理至关重要：** Zstd方法速度很快，数据压缩速度可达 80–150 MB/s，解压速度可达 150–300 MB/s。它的速度不如 Token方法，但吞吐量高于混合方法，同时仍能带来显著的压缩收益（节省 45–65% 的空间）。

• **不希望产生分词开销：** 有些应用程序可能希望完全避免分词开销，原因可能是许可问题、依赖项管理或性能需求。Zstd方法无需分词库即可有效压缩文件，因此很适合这些情况。

表 4：按用例分类的推荐压缩策略

内容类型	**方法**	**主要依据**
文档	令牌	针对标准散文进行了优化，并适用于顺序文本流。
技术文档	混合	同时捕捉自然语言和结构性样板文本。
代码 / JSON	混合	较高的语义冗余可实现最大压缩比。
多语言	Zstd	在不同场景下保持稳定的性能字符编码。
实时 API	Zstd	优先考虑吞吐量，以尽量减少系统延迟。
归档日志	混合	最大限度地节省空间，适用于大规模存储。

这些用例代表了基于大语言模型的应用中最常见的运营领域，在这些领域中，提示管理至关重要。虽然上述建议表明，内容类型与压缩方法之间应实现最佳匹配，但**LoPace**架构有意保持灵活；这三种策略可以互换，并可根据最终用户对延迟和存储约束的独特权衡，调整以适应任何特定用例。

6.2 生产部署注意事项

6.2.1 配置调优

Zstd 压缩级别是一个可调参数，可根据性能要求进行调整。我们根据实验中观察到的压缩率与处理速度之间的权衡，以及 Zstd 的设计，将 Zstd 级别范围（1–22）划分为三个档次：

• **级别 1–5：**实时应用，低延迟要求。这些级别优先考虑速度而非压缩率：压缩和解压缩速度最快，因此适用于提示

存储或检索必须在严格的延迟预算内完成的场景（例如交互式 API、实时流水线）。我们选择此范围，是因为对于典型大小的提示，Zstd 级别 1–5 的计算成本最低，处理时间不到一毫秒。

• **级别 10–15：**均衡性能（推荐默认值）。该层级提供了实用的折中：在保持适用于交互式和批处理工作负载的吞吐量的同时，可达到级别 22 所能获得的最大压缩率的约 90–95%。我们将级别 15 作为默认值（如第4.5节所述），因为它能实现这种平衡，而不会产生级别 19–22 那样陡增的时间成本。

• **级别 19–22：**最大压缩，批处理。这些级别以速度为代价最大化压缩率；当存储成本占主导且提示仅压缩一次时（例如归档、离线索引），它们最为合适。我们选择 19–22，是因为超过级别 15 后，压缩率的边际收益递减，而处理时间显著增加，因此该范围保留给延迟次于空间节省的场景。

自适应方法选择：在仅词元压缩相对于原始 UTF-8 会使数据膨胀的场景中（例如需要 uint32 打包的以 ASCII 为主的文本），系统可以自动绕过仅词元模式并回退到仅 Zstd 压缩。未来工作应根据估计的压缩率、令牌 ID 分布和提示词特征实现此类自适应选择，以自动优化压缩效率。

6.2.2 跨实例兼容性

跨实例兼容性是 LoPace 的一个非常重要的组成部分。只要使用相同的分词器模型，就可以在不同实例上执行压缩和解压缩：

$$C_1.\text{compress}(T) \rightarrow C_2.\text{decompress}() = T \tag{40}$$

这使以下功能成为
可能：

- 分布式压缩/解压缩
- 数据库存储与应用层解压缩
- 微服务架构
- 负载均衡

6.2.3 数据库集成

混合方法为数据库存储带来以下优势：

1. **可搜索性：** 无需完全解压缩即可对词元 ID 建立索引并进行搜索，从而能够高效查询压缩数据 2. **一致性：** 固定的分词器可确保相似提示的压缩率稳定 3. **效率：** 为数百万条记录节省最大空间，降低存储成本并提升查询性能

与数据库原生压缩的关系：数据库系统提供多个层级的原生压缩：(a) 页级压缩，即在存储前压缩数据库页；(b) 针对特定数据类型优化的列式压缩方案；以及 (c) 用于管理领域特定模式的应用层词典管理。LoPace 运行于

应用层，在数据库插入前压缩数据。这种方法能够对压缩设置进行细粒度控制，并针对提示的特定特征进行优化，但并不能取代数据库原生压缩。实际上，经过 LoPace 压缩的提示还可以由数据库系统进一步压缩，不过额外收益可能微乎其微。未来的工作应评估应用层压缩与数据库原生压缩之间的相互作用，以确定最佳部署策略。

7 性能基准测试

本节全面总结了各种压缩方法和提示大小的性能基准测试结果，汇总了第4节中的详细分析。

7.1 各方法的压缩率

表5：各方法的压缩率（386 个提示）

方法	平均值	Min	Max
Zstd	4.76x	1.22x	19.77x
令牌	1.02x	0.74x	2.05x
混合	4.89x	1.22x	19.09x

压缩率表展示了全部 386 个评估提示的性能。混合方法始终达到最高的平均压缩率（4.89x），且提示越长，优势越明显。这种尺寸相关性能表明，较长的文本为模式识别和冗余消除提供了更多机会，对于冗余度较高的内容，最大压缩率可达 19.09x。

7.2 按方法划分的空间节省

表6：按方法划分的空间节省（386 个提示）

方法	Mean	Min	Max
Zstd	70.2%	17.8%	94.9%
令牌	1.4%	-36.0%	51.2%
混合	72.2%	18.1%	94.8%

空间节省可直接转化为存储成本降低和基础设施效率提升。混合方法实现了72.2%的平均空间节省，对于高度冗余的提示词，最大节省幅度可达94.8%，根据内容特征将存储需求降低3.5–18.5倍，对大规模部署具有显著影响。

7.3 吞吐量（MB/s）

表7：各方法的吞吐量表现（MB/s，386 个提示）

方法	压缩（平均值）	解压缩（平均值）
Zstd	10.7	132.9
令牌	4.6	8.5
混合	3.3	2.3

吞吐量测量表明，所有方法均达到生产就绪性能，且解压缩性能始终优于压缩性能。Zstd方法实现了最高的压缩吞吐量（平均10.7 MB/s），非常适合需要快速压缩的低延迟应用。混合方法虽然实现了最佳压缩率，但由于采用两阶段处理，吞吐量较低（平均3.3 MB/s），体现了压缩效率与处理速度之间的权衡。

8 结论

本文介绍了 LoPace，这是一款用于大语言模型应用中高效提示存储的压缩引擎。我们证明，三种压缩方法——基于 Zstandard 的方法、基于令牌的方法和混合方法——在保持 100% 无损重建的同时，显著提升了存储效率。

8.1 主要贡献

1. **新颖的混合方法：** 将分词与 Zstd 压缩相结合，获得了优于单独使用任一方法的压缩率（平均值为 4.89x，范围为 1.22–19.09x），展现了结合语义压缩技术与序列压缩技术所产生的乘数效应。2. **满足生产需求的性能：** 全面的基准测试表明，压缩速度可达 50–200 MB/s，且内存占用极小（低于 10 MB），因此 LoPace 适用于生产部署。3. **数学保证：** 形式化证明与经验验证证实，所有方法均可实现完整的无损重建。4. **全面评估：** 通过对多项指标（压缩率、空间节省、吞吐量和内存）进行深入分析，为部署决策提供了切实可行的参考。

8.2 影响与应用程序

LoPace 解决了大语言模型应用部署中的关键挑战：

- **成本降低：** 存储缩减 70–80%，为大规模部署显著节省成本
- **性能提升：** 数据库大小缩减可提升查询性能和系统响应能力
- **可扩展性：** 高效的扩展特性使系统能够持续增长，而无需按比例增加基础设施
- **灵活性：**多种压缩方法可针对特定使用场景进行优化

8.3 总结

LoPace 为大语言模型应用部署中的现实挑战提供了实用解决方案。其理论基础、全面评估和可用于生产环境的实现，使其成为从事大规模大语言模型系统开发的开发者和组织的宝贵工具。

LoPace 已开源，并提供了详尽的文档和示例，使研究与开发社区能够在此基础上继续开展工作，进一步推进提示压缩技术。

8.4 局限性与未来工作

8.4.1 当前局限性

1. **分词器依赖与版本管理：** 解压缩需要匹配的分词器模型。这会带来长期脆弱性问题：(a) 分词器版本变更（例如词汇表更新、模型转换）可能导致先前压缩的数据不再兼容；(b) 不同的大语言模型使用不同的分词器，限制了跨模型兼容性；(c) 多语言提示可能无法使用英语优化分词器进行高效分词。我们建议将分词器元数据（模型标识符、版本）与压缩负载一同存储，以便进行迁移和兼容性检查。制定迁移与兼容性计划至关重要，可避免随时间推移出现数据锁定或损坏风险。我们也可以选择使用原生 Zstd 方法进行压缩，从而在保持高吞吐量的同时实现良好的压缩效果。

2. **定宽编码的局限性：** 我们的二进制打包策略采用定宽编码（uint16/uint32），而非变长整数编码（VByte/LEB128）或位打包方案。这一设计选择未能充分探索潜在的效率提升，尤其是在仅词元压缩中，对于包含许多较小词元 ID 的序列，变长编码可以减少存储开销。此外，对于 cl100k_base 分词器，许多实际提示都需要使用 uint32 打包（每个词元 4 字节），因此对于以 ASCII 为主的文本，如果不使用 Zstd 进行后续压缩，仅词元压缩后的数据可能会比原始 UTF-8 更大。

3. **基线对比有限：** 我们的评估未纳入与若干重要基线方法的比较：(a) 使用代表性提示语料库构建训练词典的 Zstd，(b) 不同质量级别的 Brotli，(c) gzip/DEFLATE，以及 (d) 其他混合级联方案（例如 Brotli→Zstd、Zstd→LZ4HC）。Zstd 字典训练是领域特定文本压缩中广为人知且非常有效的技术，应将其作为强有力的基线进行评估。加入这些比较将有助于强化我们的分析，并更好地展现 LoPace 相对于现有方法的定位。

4. **处理开销：** 由于采用两阶段处理（分词 + Zstd），混合方法会带来更高的计算成本，导致其吞吐量低于单独使用各方法的情况。这种权衡更注重压缩率而非速度。

5. **短提示的格式开销：** 非常短的提示（少于 1,000 个字符）可能因格式开销（格式字节、压缩元数据）而获得较低的压缩率，因为这些固定成本在总压缩大小中所占的比例更大。然而，我们的评估表明，即使是最短的提示（129 个字符），压缩率也超过 1.2 倍。

6. **多语言和特定领域的局限性：** 我们的评估主要关注英语文本。此方法可能无法很好地推广到以下场景：(a) 多语言提示，其中针对英语优化的分词器可能效率不高；(b) 具有

不同压缩特性的代码密集型提示；(c) JSON/元数据含量较高的对话日志；以及 (d) 可能无法被高效分词的领域专用术语。需要按领域进行评估，以衡量这些场景。

7. **流式处理与分块：** 我们没有评估超长提示或对话历史的流式处理或分块操作。不同分块大小在性能与压缩率之间的权衡仍有待探索。

8.4.2 未来研究方向

1. **变长整数编码：** 评估用于令牌 ID 编码的 VByte、LEB128 和位打包方案，以提高仅词元压缩效率，尤其适用于包含大量较小词元 ID 的序列。

2. **Zstd 字典训练：** 针对具有代表性的提示语料库开发并评估 Zstd 字典训练，并将压缩率和吞吐量与我们的混合方法进行比较。这有助于确定特定领域的词典能否达到或超越混合方法的性能。

3. **全面的基准比较：** 在多种提示语料库上，与 Brotli（不同质量级别）、gzip/DEFLATE 以及其他混合级联方案（例如 Brotli→Zstd、Zstd→LZ4HC）进行全面比较。

4. **自适应压缩：** 根据提示特征（大小、内容类型、令牌 ID 分布）开发动态方法选择机制，以自动选择最优压缩策略。

5. **分词器版本管理与迁移：** 设计稳健的分词器元数据存储和迁移策略，以应对分词器版本变更、模型转换以及跨模型兼容性。

6. **大规模评估：** 将评估扩展至数千或数百万条提示，覆盖多样化领域（代码、JSON 日志、多语言内容、聊天记录），并开展统计分析和按领域报告。

7. **系统级内存分析：** 使用系统级性能分析工具（RSS/USS 指标、原生内存分配跟踪），以准确测量内存使用量。

8. **全面吞吐量分析：** 进行详细的吞吐量测量，并报告硬件规格、CPU 利用率指标、并发分析及统计波动情况。

9. **流式处理与分块：** 评估针对超长提示的流式处理和分块操作，分析不同块大小下的性能与压缩比权衡。

10. **令牌流存储模式：** 探索一种仅存储令牌流的模式（可在 Zstd 压缩后存储），以避免在下游大语言模型推理流水线中反复进行去分词和重新分词，因为推理最终需要的是词元。

11. **并行处理：** 实现多线程压缩，以便通过批量操作提升多核系统上的吞吐量。

12. **硬件加速：** 研究 GPU 加速的分词与压缩，以及 CPU 向量化（例如词典训练、批处理），以提升吞吐量。

13. **整数压缩方案：** 评估令牌 ID 流的熵编码，以及轻量级整数压缩器（例如增量编码），以利用令牌序列中的序列模式。14. **多语言与领域专用优化：** 针对多语言提示、代码密集型内容和领域专用术语，开发专门的压缩策略。15. **有损压缩探索：** 研究特定使用场景下可接受的损失情形，即不要求完美重建的情况。

致谢

衷心感谢开源社区为我们提供所需的库，使 Zstandard 和 Tiktoken 得以实现。我们还要感谢构建并维护 Python 科学计算环境的人们，感谢他们提供的工具和框架，让这项研究得以完成。

您可以在https://github.com/connectaman/LoPace找到 LoPace 的源代码、文档及其他材料。

参考文献

[1] Shannon, C. E. (1948). 通信的数学理论。贝尔系统技术期刊，27(3), 379–423.[2] Ziv, J., & Lempel, A. (1977). 用于序列数据压缩的通用算法。*IEEE* 信息论汇刊，23(3), 337–343. [3] Ziv, J., & Lempel, A. (1978). 通过变速率编码压缩单个序列。*IEEE* 信息论汇刊，24(5), 530–536.[4] Huffman, D. A. (1952). 一种构造最小冗余码的方法。*IRE* 会议录，40(9), 1098–1101.[5] Cleary, J. G., & Witten, I. H. (1984). 使用自适应编码和部分字符串匹配进行数据压缩。*IEEE* 通信汇刊，32(4), 396–402.[6] Deutsch, P. (1996). DEFLATE 压缩数据格式规范第 1.3 版。*RFC1951*。[7] Collet, Y. (2016). Zstandard 压缩及 'application/zstd' 媒体类型。*RFC8878*。[8] Sennrich, R., Haddow, B., & Birch, A. (2016). 使用子词单元翻译罕见词的神经机器翻译。自然语言处理中的经验方法协会第 *54* 届年会会议录 *(ACL)*，1715–1725.[9] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). 语言模型是无监督多任务学习者。*OpenAI* 博客。[10]Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT：用于语言理解的深度双向变换器预训练。*NAACL-HLT* 会议录，4171–4186.[11] Gori, M., & Lippi, M. (2015). 时间序列压缩：综述。国际数据挖掘、建模与管理杂志，7(1), 1–23。

[12] Burrows, M., & Wheeler, D. J. (1994). 一种基于块排序的无损数据压缩算法。数字设备公司技术报告，124.[13] Li, S., & Li, K. (2013). 多核处理器上的并行无损数据压缩。并行与分布式计算杂志，73(5), 608–620.[14] Jiang, H., Wang, D., & Dou, Q. (2023). 大语言模型的提示压缩。*arXiv* 预印本 *arXiv:2305.11147*。[15] Ge, T., & Wei, F. (2023). 高效大语言模型推理的提示缓存。*2023*年自然语言处理中的经验方法会议会议录。