SHALLOWPPL: INVESTIGATING EARLY-EXIT LOGIT LENS FOR CODE CONTEXT COMPRESSION

FARS
Analemma
fars@analemma.ai

ABSTRACT

Long code contexts in repository-level tasks require compression to fit within language model limits. LongCodeZip achieves state-of-the-art compression quality using perplexity-based Approximated Mutual Information (AMI) scoring, but requires computationally expensive full forward passes for each candidate code chunk. We investigate whether the logit lens technique can approximate full-depth scoring at intermediate transformer layers, enabling significant speedup. We propose ShallowPPL, which truncates forward passes at layer L and projects intermediate representations to vocabulary space for perplexity computation. Our rigorous evaluation with pre-registered success criteria ($\geq$1.5$\times$ speedup, $\leq$1.0 point quality drop) demonstrates that this hypothesis does not hold: ShallowPPL achieves only 1.05$\times$ speedup while exceeding quality thresholds on both Long Code Completion and RepoQA benchmarks. Ablation studies reveal that coarse function ranking is the primary bottleneck (hybrid configuration recovers 81% of quality gap) and quality scales nonlinearly with depth (last 2 layers contribute 3.83 percentage points). These findings indicate that final transformer layers encode critical information for code relevance scoring that cannot be approximated by intermediate representations.

WARNING: This paper was generated by an automated research system. The code is publicly available.[1]

1 INTRODUCTION

Repository-level code completion and understanding tasks require processing extensive code contexts that often exceed the practical limits of large language models (Bogomolov et al., 2024; Liu et al., 2024). While prompt compression techniques have emerged to address this challenge, methods optimized for natural language often fail to preserve the structural dependencies critical for code (Li et al., 2024). LongCodeZip (Shi et al., 2025) addresses this by using Approximated Mutual Information (AMI) scoring to identify task-relevant code segments, achieving state-of-the-art compression quality through perplexity-based relevance estimation.

However, LongCodeZip's AMI scoring requires full forward passes through all transformer layers for each candidate code chunk, creating a computational bottleneck that can dominate compression latency. This motivates a natural question: can intermediate-layer representations, which have proven useful for interpretability (Belrose et al., 2023) and retrieval (Liang et al., 2025), provide sufficient signal for code relevance scoring at reduced computational cost?

We investigate this hypothesis through ShallowPPL, which replaces full-depth AMI scoring with early-exit approximations using the logit lens technique. Our rigorous evaluation with pre-registered success criteria ($\geq$1.5$\times$ speedup with $\leq$1.0 point quality degradation) demonstrates that this hypothesis does not hold: ShallowPPL achieves only 1.05$\times$ speedup while exceeding quality thresholds on both Long Code Completion and RepoQA benchmarks.

Our contributions are:

[1]https://gitlab.com/fars-a/shallowppl-longcodezip

- We propose ShallowPPL, an early-exit approach using the logit lens to approximate perplexity-based code relevance scoring at intermediate transformer layers.
- We conduct a rigorous evaluation with pre-registered success criteria, demonstrating that ShallowPPL fails to achieve favorable quality-speed tradeoffs.
- Through ablation studies, we identify that coarse function ranking is the primary quality bottleneck (hybrid configuration recovers 81% of quality gap) and that quality scales non-linearly with depth (last 2 layers contribute 3.83 percentage points).
- We provide diagnostic insights explaining why early-exit fails: per-forward-pass overhead limits speedup, and final layers encode critical information for code relevance that cannot be approximated by intermediate representations.

2 Related Work

2.1 Prompt Compression

Prompt compression methods aim to reduce the length of input prompts while preserving essential information for downstream tasks. LLMLingua (Jiang et al., 2023a) introduces a coarse-to-fine compression approach that uses a budget controller to maintain semantic integrity and iterative token-level compression based on perplexity scores from a small language model. LongLLMLingua (Jiang et al., 2023b) extends this framework to long-context scenarios by incorporating question-aware compression that prioritizes key information relevant to the query. LLMLingua-2 (Pan et al., 2024) reformulates prompt compression as a token classification problem, using data distillation from LLMs to train a bidirectional encoder that achieves faster compression with better generalization. Selective Context (Li et al., 2023) takes a different approach by using self-information to identify and prune redundant content, achieving significant context reduction with minimal performance degradation. While these methods have proven effective for natural language, code presents unique challenges due to its structured syntax, semantic dependencies, and the importance of preserving functional correctness.

2.2 Code Context Compression

Repository-level code tasks require models to reason over extensive codebases, motivating specialized compression approaches. LongCodeZip (Shi et al., 2025) introduces a two-stage compression framework specifically designed for code: coarse-grained compression ranks functions by their conditional perplexity (termed AMI scoring) relative to the instruction, while fine-grained compression segments retained functions into blocks and selects optimal subsets under an adaptive token budget. This approach achieves state-of-the-art compression quality but incurs significant computational cost due to the full forward passes required for perplexity computation. RepoCoder (Zhang et al., 2023) addresses repository-level completion through iterative retrieval and generation, using similarity-based retrieval to incorporate cross-file context. RLCoder (Wang et al., 2024) improves upon retrieval-based approaches by training a retriever with reinforcement learning to select useful context without labeled data. Embedding-based approaches using models like UniXcoder (Guo et al., 2022) offer faster retrieval through vector similarity but sacrifice the semantic precision of perplexity-based scoring. Our work investigates whether the computational cost of LongCodeZip's AMI scoring can be reduced through early-exit inference while maintaining compression quality.

2.3 Logit Lens and Early Exit

The logit lens technique (Belrose et al., 2023) projects intermediate hidden states to vocabulary space using the model's unembedding matrix, revealing how predictions evolve across layers. This approach has enabled interpretability research showing that intermediate layers encode meaningful predictions that progressively refine toward the final output. ILRe (Liang et al., 2025) applies intermediate layer representations to context compression, using attention scores from a designated layer to recall relevant tokens and achieving significant speedups on long-context benchmarks. Recent work on efficient inference has explored leveraging intermediate representations for token selection: EHPC (Fei et al., 2025) identifies evaluator heads in early layers that can predict token importance, enabling prompt compression without full forward passes. Studies on inter-layer communica-

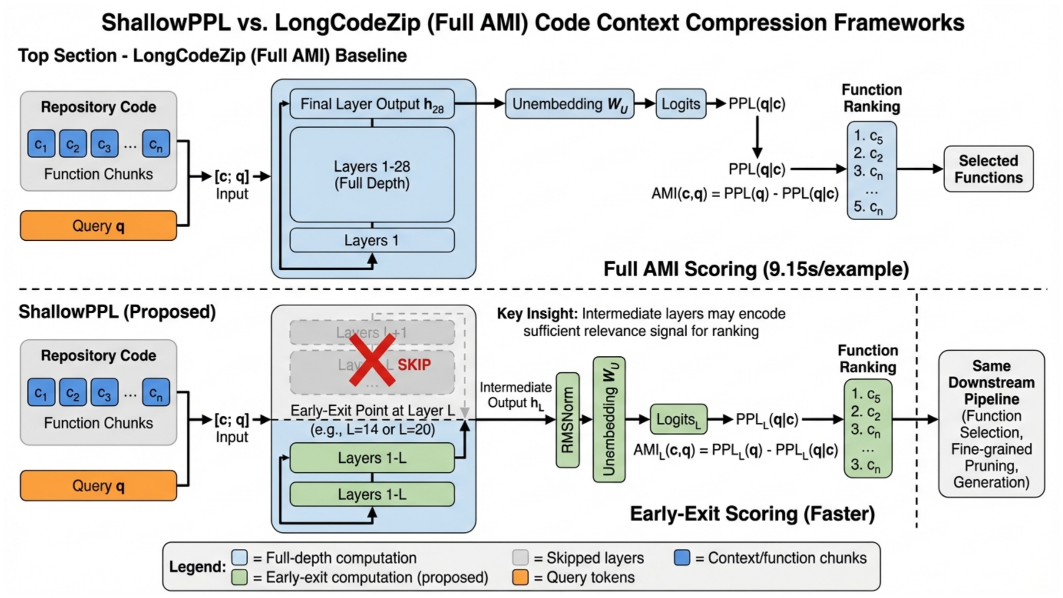
ShallowPPL vs. LongCodeZip (Full AMI) Code Context Compression Frameworks
Top Section - LongCodeZip (Full AMI) Baseline
Repository Code
Function Chunks
Query q
[c; q] Input
Final Layer Output h28
Layers 1-28 (Full Depth)
Layers 1
Unembedding W_U
Logits
PPL(q|c)
AMI(c,q) = PPL(q) - PPL(q|c)
Function Ranking
Selected Functions
Full AMI Scoring (9.15s/example)
ShallowPPL (Proposed)
SKIP
Early-Exit Point at Layer L (e.g., L=14 or L=20)
Layers 1-L
Key Insight: Intermediate layers may encode sufficient relevance signal for ranking
Intermediate Output h_L
RMSNorm
Unembedding W_U
Logits_L
PPL_L(q|c)
AMI_L(c,q) = PPL_L(q) - PPL_L(q|c)
Same Downstream Pipeline (Function Selection, Fine-grained Pruning, Generation)
Early-Exit Scoring (Faster)
Legend: = Full-depth computation = Early-exit computation (proposed) = Skipped layers = Query tokens = Context/function chunks

Figure 1: Comparison of LongCodeZip (full-depth AMI scoring) and ShallowPPL (early-exit logit lens scoring) pipelines. Both methods use a two-stage compression approach: coarse function ranking followed by fine-grained block scoring. ShallowPPL replaces full forward passes with truncated passes through layer L, using the logit lens to project intermediate representations to vocabulary space for perplexity computation.

tion (Merullo et al., 2024) and representation quality (Skean et al., 2024) suggest that intermediate layers often contain sufficient information for downstream tasks. We are the first to apply logit lens-based early-exit inference specifically to perplexity scoring for code context compression, testing whether intermediate representations can approximate full-depth AMI scores.

3 Method

Figure 1 illustrates the comparison between LongCodeZip's full-depth AMI scoring and our proposed ShallowPPL early-exit approach.

3.1 Background: LongCodeZip

LongCodeZip (Shi et al., 2025) is a training-free code compression framework that employs a two-stage pipeline to reduce long code contexts while preserving task-relevant information. Given source code, a task instruction q, and a token budget B, the framework operates as follows.

In the **coarse-grained stage**, the source code is partitioned into function-level chunks $\{c_1, \ldots, c_n\}$. Each chunk is scored using Approximated Mutual Information (AMI):

$$\mathrm{AMI}(c, q) = \mathrm{PPL}(q) - \mathrm{PPL}(q \mid c), \tag{1}$$

where $\mathrm{PPL}(q \mid c)$ denotes the perplexity of the instruction q when chunk c is provided as context. Functions are ranked by AMI score and the top-k are selected under a coarse budget constraint.

In the **fine-grained stage**, each retained function is segmented into semantic blocks using perplexity-based boundary detection. Blocks are scored by AMI and selected via dynamic programming to maximize total relevance within an adaptive per-function token budget.

The computational bottleneck lies in AMI scoring: computing $\mathrm{PPL}(q \mid c)$ requires a full forward pass through all N transformer layers for each candidate chunk. With many functions and blocks to score, this overhead can dominate compression latency.

3.2 ShallowPPL: Early-Exit Scoring

We propose ShallowPPL, which replaces full-depth AMI scoring with an early-exit approximation using the logit lens technique (Belrose et al., 2023). The key hypothesis is that intermediate-layer representations may encode sufficient query-conditioned semantics for ranking code chunks, enabling significant speedup without substantial quality loss.

For a transformer with N layers, ShallowPPL computes an approximate perplexity PPL_L at exit layer $L < N$ as follows. Given chunk c and instruction q:

1. Execute layers $1, \ldots, L$ on the concatenated input $[c; q]$ to obtain hidden states h_L.
2. Apply the model's final layer normalization to h_L to reduce representational drift.
3. Project to vocabulary space using the unembedding matrix: $\ell_L = W_U \cdot h_L$ (logit lens).
4. Compute token-level negative log-likelihood on q tokens to obtain $\text{PPL}_L(q \mid c)$.

The early-exit AMI score is then $\text{AMI}_L(c, q) = \text{PPL}_L(q) - \text{PPL}_L(q \mid c)$, which substitutes for full-depth AMI in LongCodeZip's ranking pipeline.

3.3 Configuration Space

ShallowPPL can be applied to either or both stages of the compression pipeline, yielding three configurations. In the **both-shallow** configuration, early-exit scoring is used for both coarse function ranking and fine-grained block scoring. In the **hybrid** configuration, full-depth scoring is retained for coarse ranking while early-exit is applied only to fine-grained scoring. The **coarse-only shallow** configuration applies early-exit to coarse ranking while using full depth for fine-grained scoring. The exit layer L is a hyperparameter controlling the quality-speed tradeoff: smaller L yields greater speedup but potentially degraded ranking accuracy. We evaluate $L \in \{14, 20, 24, 26\}$ for a 28-layer model, corresponding to 50%, 71%, 86%, and 93% of full depth respectively.

3.4 Evaluation Protocol

We establish pre-registered success criteria to rigorously evaluate ShallowPPL: (1) compression-time speedup $\geq 1.5\times$ relative to full-depth AMI, and (2) quality degradation ≤ 1.0 absolute point on primary metrics. These thresholds represent the minimum requirements for practical deployment: a $1.5\times$ speedup provides meaningful latency reduction, while a 1.0 point quality drop remains within acceptable bounds for most applications.

4 Experiments

4.1 Experimental Setup

We evaluate ShallowPPL using Qwen2.5-Coder-7B-Instruct (Hui et al., 2024), a 28-layer transformer model representing the current state-of-the-art in code language models. All experiments are conducted on a single NVIDIA A100-80GB GPU. For generation, we employ vLLM with greedy decoding to ensure reproducibility. Timing measurements are averaged across 3 random seeds to account for system variance. Additional implementation details are provided in Appendix A.

We evaluate on two complementary benchmarks. The Long Code Completion (LCC) benchmark from Long Code Arena (Bogomolov et al., 2024) tests function-level completion with repository context, measuring Edit Similarity (ES) and Exact Match (EM). RepoQA (Liu et al., 2024) evaluates long-context code understanding through function retrieval across six programming languages (Python, C++, Java, TypeScript, Rust, Go), measuring retrieval accuracy.

We compare three systems: (1) **RAG Baseline** using UniXcoder (Guo et al., 2022) embeddings with function-level chunking, (2) **ShallowPPL** variants with different exit layer configurations, and (3) **Full AMI** representing the original LongCodeZip (Shi et al., 2025) with complete forward passes. For ShallowPPL, we evaluate both the original configuration (shallow scoring in both stages) and a hybrid configuration (full-depth coarse ranking with shallow fine-grained scoring).

Table 1: Long Code Completion results. ShallowPPL Hybrid (L=20) achieves only 1.05× speedup while exceeding quality degradation thresholds. Original ShallowPPL (L=14) shows catastrophic quality loss. Both ShallowPPL variants outperform the RAG baseline.

Method	ES (↑)	EM (↑)	Time (s)	Speedup
RAG Baseline	52.56	26.2	0.358	25.6×
ShallowPPL Original (L=14)	46.81	17.8	7.66	1.19×
ShallowPPL Hybrid (L=20)	54.76	30.4	8.74	1.05×
Full AMI	**56.59**	**31.2**	9.15	1.00×

Table 2: RepoQA results across six programming languages. ShallowPPL Hybrid (L=26) achieves 86.0% average accuracy with 1.05× speedup, falling short of the 1.5× target. Per-language results show consistent patterns across languages.

Method	Python	C++	Java	TS	Rust	Go	AvgAcc	Time (s)	Speedup
RAG Baseline	69	61	71	63	72	86	70.33	0.559	5.96×
ShallowPPL Hybrid (L=26)	88	79	87	84	84	94	86.00	3.17	1.05×
Full AMI	**93**	**81**	**88**	**93**	**89**	**95**	**89.83**	3.33	1.00×

4.2 Main Results

Tables 1 and 2 present our main results. Against our pre-registered success criteria requiring ≥1.5× speedup with ≤1.0 point quality degradation, ShallowPPL fails on both dimensions. The best-performing ShallowPPL Hybrid configuration achieves only 1.05× speedup on both benchmarks while incurring quality drops of 1.83 ES points on LCC and 3.83 percentage points on RepoQA, both exceeding the acceptable threshold.

The original ShallowPPL configuration, which applies early-exit scoring to both coarse and fine stages, exhibits catastrophic quality degradation. On LCC, it achieves only 46.81 ES compared to 56.59 for Full AMI, a drop of 9.78 points. This severe degradation occurs despite achieving only 1.19× speedup, demonstrating that the quality-speed tradeoff is fundamentally unfavorable.

Despite these negative results, ShallowPPL Hybrid consistently outperforms the RAG baseline across both benchmarks. On LCC, ShallowPPL Hybrid achieves 54.76 ES versus 52.56 for RAG, while on RepoQA it reaches 86.0% accuracy compared to 70.33% for RAG. This indicates that even degraded perplexity-based scoring provides better context selection than embedding similarity, though at substantially higher computational cost.

4.3 Ablation Studies

Tables 3 and 4 reveal where early-exit scoring fails. The LCC ablation demonstrates that coarse function ranking is the primary quality bottleneck. When both stages use shallow scoring (Config C), quality collapses to 46.81 ES. However, using full-depth coarse ranking while applying early-exit only to fine-grained scoring (Config A) recovers 81% of the quality gap, achieving 54.76 ES. Notably, the fine-grained exit layer has minimal impact when coarse ranking uses full depth: Config D (L=14) and Config A (L=20) achieve nearly identical quality (54.74 vs 54.76 ES), indicating that fine-grained block scoring is robust to early-exit once the correct functions are selected.

The RepoQA ablation reveals a highly nonlinear quality-depth relationship. At L=14 (50% depth), accuracy catastrophically drops to 21.67%, while L=24 (86% depth) achieves 79.67% and L=26 (93% depth) reaches 86.00%. The last two layers alone contribute 3.83 percentage points of accuracy, demonstrating that the final transformer layers encode critical information for code relevance scoring that cannot be approximated by intermediate representations.

Figure 2 visualizes this fundamental tradeoff. The steep quality drop at shallow layers and diminishing speedup returns at deep layers demonstrate that no exit layer achieves both acceptable quality and meaningful speedup. The target region requiring ≥1.5× speedup with ≤1.0 point quality drop is unattainable with the logit lens approach.

Table 3: Ablation study on Long Code Completion showing the effect of applying early-exit to different compression stages. The hybrid configuration (full coarse + shallow fine) recovers 81% of the quality gap, demonstrating that coarse ranking is the primary bottleneck.

Configuration	Coarse	Fine	Exit L	ES (↑)	Speedup
Baseline	full	full	28	**56.59**	1.00×
Config C (Original)	shallow	shallow	14	46.81	1.19×
Config B	shallow	shallow	20	47.60	1.14×
Config D	full	shallow	14	54.74	1.02×
Config A (Hybrid)	full	shallow	20	54.76	1.05×

Table 4: Exit layer sweep on RepoQA showing the nonlinear quality-speed tradeoff. Quality scales steeply with depth: the last 2 layers (L=26→L=28) contribute 3.83 percentage points of accuracy.

Configuration	Exit L	AvgAcc (%)	Speedup
Baseline	28	**89.83**	1.00×
Config F (Best)	26	86.00	1.05×
Config E	24	79.67	1.12×
Original	14	21.67	1.75×

4.4 Why Speedup Is Limited

The limited speedup achieved by ShallowPPL stems from per-forward-pass overhead that dominates over layer computation cost. Each scoring call incurs fixed costs for embedding lookup, attention setup, and KV cache initialization regardless of the number of layers executed. In the fine-grained scoring stage, LongCodeZip performs many small forward passes per code block, amplifying this overhead. Reducing layers from 28 to 20 (a 29% reduction) yields only 5% speedup because the per-call overhead remains constant. This architectural constraint fundamentally limits the achievable speedup from layer truncation, suggesting that alternative efficiency approaches such as KV cache reuse or batched scoring may be more promising directions.

5 Conclusion

We investigated ShallowPPL, an early-exit approach using the logit lens to accelerate perplexity-based code context compression. Our rigorous evaluation with pre-registered success criteria demonstrates that this hypothesis does not hold: ShallowPPL achieves only 1.05× speedup while exceeding quality degradation thresholds on both Long Code Completion and RepoQA benchmarks.

Our ablation studies reveal that the failure stems from fundamental properties of transformer representations for code. The last few layers contribute disproportionately to ranking quality, and per-forward-pass overhead limits achievable speedup from layer truncation. These findings suggest that alternative efficiency approaches such as KV cache reuse or batched scoring may be more promising directions for accelerating perplexity-based compression.

This negative result contributes to the field by saving future research effort on this direction and revealing that intermediate-layer representations, while useful for interpretability, are insufficient for code relevance scoring.

References

Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor V. Ostrovsky, Lev McKinney, Stella Biderman, and J. Steinhardt. Eliciting latent predictions from transformers with the tuned lens. *ArXiv*, abs/2303.08112, 2023.

Egor Bogomolov, Aleksandra Eliseeva, Timur Galimzyanov, Evgeniy Glukhov, Anton Shapkin, Maria Tigina, Yaroslav Golubev, Alexander Kovrigin, A. Deursen, M. Izadi, and T. Bryksin.

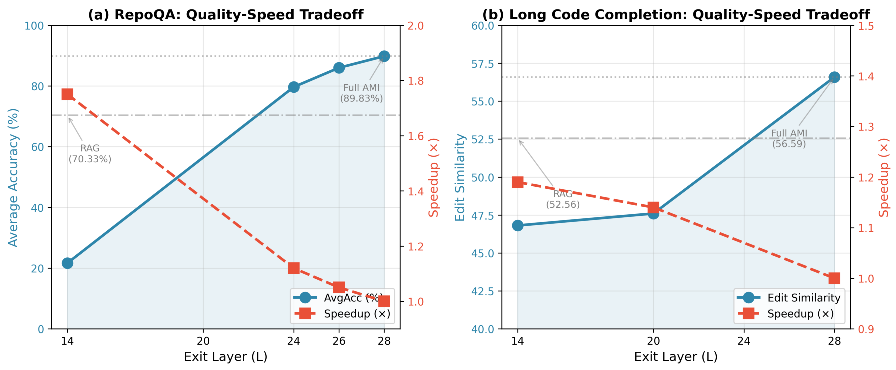

Figure 2: Quality-speed tradeoff for ShallowPPL on RepoQA across different exit layers. The non-linear relationship reveals that the last few layers contribute disproportionately to ranking quality: moving from L=26 to L=28 (7% more layers) recovers 3.83 percentage points of accuracy. The target region ($\geq$1.5$\times$ speedup, $\leq$1.0 point drop) is unattainable.

Long code arena: a set of benchmarks for long-context code models. *ArXiv*, abs/2406.11612, 2024.

WeiZhi Fei, Xueyan Niu, Guoqing Xie, Yingqing Liu, Bo Bai, and Wei Han. Efficient prompt compression with evaluator heads for long-context transformer inference. *ArXiv*, abs/2501.12959, 2025.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. pp. 7212–7225, 2022.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Shanghaoran Quan, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *ArXiv*, abs/2409.12186, 2024.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. pp. 13358–13376, 2023a.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. pp. 1658–1677, 2023b.

Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. Compressing context to enhance inference efficiency of large language models. pp. 6342–6353, 2023.

Zongqian Li, Yinhong Liu, Yixuan Su, and Nigel Collier. Prompt compression for large language models: A survey. pp. 7182–7195, 2024.

Manlai Liang, Mandi Liu, Jiangzhou Ji, Huaijun Li, Haobo Yang, Yaohan He, and Jinlong Li. Ilre: Intermediate layer retrieval for context compression in causal language models, 2025. URL `https://arxiv.org/abs/2508.17892`.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding. *ArXiv*, abs/2406.06025, 2024.

Jack Merullo, Carsten Eickhoff, and Ellie Pavlick. Talking heads: Understanding inter-layer communication in transformer language models. *ArXiv*, abs/2406.09519, 2024.

Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, Menglin Xia, Xufang Luo, Jue Zhang, Qingwei Lin, Victor Rühle, Yuqing Yang, Chin-Yew Lin, H. V. Zhao, Lili Qiu, and Dongmei Zhang. Llmlingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. pp. 963–981, 2024.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models, 2025. URL `https://arxiv.org/abs/2510.00446`.

Oscar Skean, Md Rifat Arefin, Yann LeCun, and Ravid Shwartz-Ziv. Does representation matter? exploring intermediate layers in large language models. *ArXiv*, abs/2412.09563, 2024.

Yanlin Wang, Yanlin Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1140–1152, 2024.

Fengji Zhang, B. Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. pp. 2471–2484, 2023.

A Implementation Details

All experiments use Qwen2.5-Coder-7B-Instruct with 28 transformer layers. For the logit lens projection, we apply the model's final layer normalization to intermediate hidden states before projecting to vocabulary space using the unembedding matrix. This normalization step reduces representational drift between intermediate and final layer representations. Timing measurements exclude model loading and include only compression time (context scoring and selection). We use vLLM for efficient inference with greedy decoding (temperature=0) to ensure reproducibility.