CoRoVA: Compressed Representations for Vector-Augmented Code Completion

Anonymous ACL submission

Abstract

Retrieval-augmented generation has emerged as one of the most effective approaches for code completion enhancement, especially when repository-level context is important. However, adding this extra retrieved context significantly increases sequence length, raises prefill cost, and degrades time-to-first-token (TTFT), which slows down inference – a critical limitation for interactive settings such as IDEs. In this work, we introduce CoRoVA, a framework that compresses context into compact, semantically rich representations that remain interpretable to code LLMs. This improves generation quality while reducing prompt augmentation to only a few compressed single-token vectors. Our approach requires training only a small projector module and adds negligible latency. CoRoVA targets the quality–latency Pareto frontier: it consistently improves over the no-context baseline and recovers a substantial fraction of RAG's quality gain at the cost of only a few tokens, while outperforming naive truncation and prior context-compression baselines at a matched token budget. Our experiments show that CoRoVA enables a 20-38% reduction in TTFT on completion tasks compared to uncompressed RAG.

1 Introduction

Modern code completion systems are increasingly deployed in interactive development environments (IDEs) as central productivity tools, with editors such as Windsurf[1] and Cursor[2] integrating large language models (LLMs) for single- and multi-line predictions. These capabilities come with strict latency requirements: even small delays in time-to-first-token (TTFT) break the interactive coding experience. At the same time, high-quality completions often require *repository-level context*: the correct line depends on names, types, helper functions, and conventions defined elsewhere in the codebase rather than in the local window.

Retrieval-augmented generation (RAG) (Lewis et al., 2021) naturally addresses the need for non-local context (Fig. 1a), but the additional retrieved tokens significantly increase prompt processing time and consequently TTFT, making vanilla RAG *less practical for latency-critical settings such as code completion*.

A promising remedy is context compression via embedding projection. Originally introduced in multimodal models such as Flamingo (Alayrac et al., 2022) and LLaVA (Liu et al., 2023), where a vision encoder and a lightweight projector map image embeddings into a compact token sequence for the language model, the idea was extended to textual retrieval by xRAG (Cheng et al., 2024), which showed that compressed representations can match vanilla RAG performance while dramatically reducing inference cost.

No prior work has applied embedding-projection-style compression to code completion, where the latency–quality trade-off is particularly severe. Furthermore, common training objectives (cross-entropy) are *misaligned with developer-relevant metrics* such as Exact Match (EM) and Edit Similarity (ES) (Tan et al., 2024; Wu et al., 2024), limiting the effectiveness of existing approaches. Finally, code has structural modalities beyond surface text – abstract syntax trees (ASTs), control- and data-flow graphs – that can be incorporated into retrieved embeddings to *enrich the representation*.

In this work we address all three points and introduce **Co**mpressed **R**epresentati**o**ns for **V**ector-**A**ugmented (CoRoVA) code completion. Our main contributions are:

- CoRoVA is the first method to apply embedding projection to code completion. Rather

[1] https://windsurf.com/
[2] https://cursor.com/

than matching uncompressed RAG, it targets the quality–latency Pareto frontier: it consistently improves over the base model at negligible latency overhead and 20–38% lower TTFT than full RAG, and at a matched token budget outperforms naive context truncation and learned context-compression baselines (e.g., pruning, summarization, CodePromptZip; Table 2, Appendix C.3).

- We design a composite loss combining cross-entropy, an RL-inspired term that directly optimises EM and ES, and a novel cosine-alignment term that preserves distinctions among the compressed representations; prior projection-training objectives proved insufficient for code completion.

- We study the incorporation of structural code modalities (ASTs) into the projected embeddings.

2 Related Work

Coding LLMs. StarCoder (Li et al., 2023) introduced a family of code generation models, including larger LLMs optimized for code-centric dialogue and smaller ones tailored for code completion. Trained on The Stack dataset (Kocetkov et al., 2022), these models achieved strong performance, surpassing most prior approaches on both code completion and instruction-following benchmarks. The Qwen-2.5-Coder series (Hui et al., 2024) represented another significant advancement in code-focused LLMs. Trained on a proprietary mixture of data, the models were released in sizes ranging from 0.5B to 32B parameters and were designed to support text completion, code chat, and fill-in-the-middle tasks.

Context Compression Methods. Although modern decoder-only transformers benefit from inference optimizations such as KV-caching (Pope et al., 2022) and efficient attention implementations like GQA (Ainslie et al., 2023), time per-token inference latency still grows with context size. Since Retrieval Augmented Generation (Lewis et al., 2021) retrieves information from the knowledge base and puts it into the context of language models, this increases the context size that needs to be processed and subsequently increases end-to-end latency. Consequently, *compressing retrieved context into compact, informative representations* – without sacrificing semantic relevance – is *essential for deploying RAG in practice*.

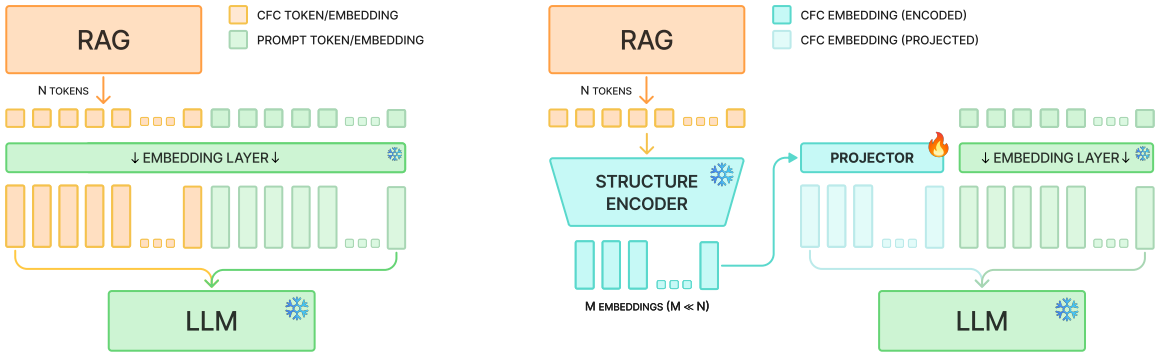
In the paper xRAG (Cheng et al., 2024) the authors propose an approach, which is similar to multi-modal language models training: they push the embedding vector of the retrieved text from textual encoder through a lightweight projector layer to align it with the reader model. The resulting architecture is trained in a two-stage manner. In the first stage, both the encoder and LLM are frozen, while the projection layer is trained with cross-entropy loss on paraphrases of the same document. During the second stage, the projector is trained on a mix of tasks such as reading comprehension, open-domain QA and summarization, adding self-distillation from RAG teacher via KL term along with usual negative log-likelihood loss.

Our approach is conceptually similar to xRAG method. By using a LLaVA-like projection from the encoder to the code completion model, we compress the retrieved context and maintain good generation quality, while lowering the TTFT. However, due to the specificity of our domain, we applied *additional techniques* to increase code-specific metrics and quality of predictions. Furthermore, we train *only the projector* with both the encoder and reader LLM frozen in a *single stage manner*.

Embedding Models for Code. Many code-search embedding models are built by taking a pretrained decoder-only language model and fine-tuning it with contrastive learning to produce useful vector representations. One such model is Qwen3-Embedding-0.6B (Zhang et al., 2025), which is based on Qwen3-0.6B (Yang et al., 2025) LLM.

Moreover, some of the encoder models were trained not only on pure text and code data, but also on structured graphs, retrieved from code: data flow graphs (DFG) and abstract syntax trees (AST). Notable examples include GraphCodeBERT (Guo et al., 2021) and UniXcoder (Guo et al., 2022), which unified code, text and graph data to improve representation quality for code-understanding and retrieval tasks.

We evaluate two encoders utilizing different code representations to understand how the choice of representation affects both projection quality and inference latency (see Sections C.2 and E).



(a) Vanilla RAG (b) CoRoVA

Figure 1: Comparison between Vanilla RAG (a) and CoRoVA (b) architectures. Instead of retrieving code passages and putting them into the context of the reader language model, CoRoVA uses a pretrained encoder to compress the code snippets and projects them into continuous tokens, thus reducing the final prompt processing time.

RL in Language Modeling. Training language models solely for next-token prediction optimizes perplexity but is misaligned with sequence-level objectives such as EM and ES. Self-Critical Sequence Training (SCST) (Rennie et al., 2017) applies REINFORCE (Williams, 1992) with a greedy-rollout baseline to directly optimize non-differentiable evaluation metrics, and we adopt it for the same reason. Alternatives such as PPO (Schulman et al., 2017) and GRPO (Shao et al., 2024) either require an additional value model or sample many rollouts per prompt, which requires more computations; SCST also achieves a variance-reduction effect but at a much lower cost.

3 Methodology

In this section, we present CoRoVA, a retrieval-augmented code completion approach that *compresses retrieved repository context into a small set of embedding vectors*, enabling the code LLM to leverage rich cross-file evidence while avoiding the token overhead and latency of vanilla RAG.

3.1 Model Architecture

To reduce the number of retrieved tokens processed by the RAG reader model, CoRoVA compresses retrieved information into a compact, task-aware representation. Specifically, we use an off-the-shelf embedding model to encode *each chunk* into a *single dense vector*. This vector is then passed through a lightweight projector to align its semantic space with the reader LLM's token embeddings.

In practice, we take the top-10 retrieved chunks for each completion target and compress them into 10 projected contextual embeddings, which are concatenated with the prompt embeddings at the LLM input (Figure 1b). This design adds negligible latency (Section 4.3 reports latency measurements), since we directly retrieve precomputed projections from the RAG database, without running the encoder or projector at the code completion time. We also experiment with fewer projected tokens ($k \leq 10$); the resulting quality–cost trade-off is reported in Appendix C.3.

Injecting the compressed tokens. Following LLaVA, we add a reserved placeholder token, `<CODE_STRUCTURE>`, to the reader vocabulary, write it once per retrieved chunk into the prompt, and overwrite the embeddings at those positions with the projected vectors. The compressed tokens thus occupy *ordinary sequence positions*: they take the reader model's standard rotary positional encodings with consecutive indices and the standard causal attention mask. We place them as one contiguous block at the very beginning of the prompt (before `<|fim_prefix|>`), as is common practice in retrieval-augmented code completion, where the retrieved cross-file context is prepended to the prompt (Zhang et al., 2023).

In our experiments, we use Qwen2.5-Coder and StarCoder2 model families as base code-completion LLMs. For encoding retrieved cross-file context, we adopt two embedding models: Qwen3-Embedding-0.6B (Zhang et al., 2025) and UniXCoder (Guo et al., 2022). We reduce each chunk to a single normalized vector by pooling the encoder's token representations: last-token (final-position) pooling for the decoder-based Qwen3-Embedding and mean pooling for the bidirectional UniXCoder. The projector follows the

LLaVA (Liu et al., 2023) projector architecture: a lightweight MLP with GELU (Hendrycks, 2016) activation and LayerNorm (Ba et al., 2016). We consider two projector architectures: a 2-layer and a 3-layer MLP. Architectural details are summarized in Table 7 and Section 4.1.

3.2 Cross-Entropy Issue

Prior projection-training work on instruction-tuned models and QA datasets is typically two-stage and optimises pure cross-entropy (CE) (Jaegle et al., 2021; Liu et al., 2023; Zemskova and Yudin, 2025), with notable exceptions: xRAG (Cheng et al., 2024) adds KL divergence, and Flamingo (Alayrac et al., 2022) uses the contrastive loss of Radford et al. (2021). CoRoVA instead trains the projector in a *single stage* (a pre-training attempt brought no gain, Appendix B.3).

CE is the standard objective for autoregressive LMs, $\mathcal{L}_{CE}(\theta) = -\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(y_t \mid y_{<t})$, but our loss ablation (Table 1) shows that CE alone is *insufficient for repository-level code completion*: it correlates weakly with EM and ES. The misalignment arises because CE is a token-level objective and does not model sequence-level correctness or error propagation: an early mistake may still lead to a fluent, low-CE continuation that is semantically wrong at the repository level. These observations motivate *directly optimising sequence-level metrics*, e.g. via reinforcement learning.

3.3 RL Loss Component

Teacher-forced maximum-likelihood training introduces exposure bias (Ranzato et al., 2015; Bengio et al., 2015): at test time the model conditions on its own previous predictions rather than the ground-truth prefix, so errors accumulate. This particularly hurts sequence-level metrics such as EM and ES. Ranzato et al. (2015) and Rennie et al. (2017) apply REINFORCE (Williams, 1992) to directly optimise such non-differentiable, sequence-level metrics. With reward $r(y) = \mathrm{EM}(y) + \mathrm{ES}(y)$, the policy gradient is

$$\nabla_\theta \mathcal{L}_R(\theta) = -\mathbb{E}_{y\sim p_\theta}\big[r(y)\,\nabla_\theta \log p_\theta(y)\big], \quad (1)$$

which is unbiased but exhibits high variance when approximated by a single sample.

To reduce that variance we adopt Self-Critical Sequence Training (SCST) (Rennie et al., 2017), which subtracts the reward of a greedy rollout as a baseline. Concretely, at each training step we draw a trajectory $y^s \sim p_\theta(\cdot)$ from the current policy and compute the greedy trajectory $y^g = \arg\max_y p_\theta(y)$ deterministically; the SCST estimator is then

$$\mathcal{L}_R(\theta) = -\big(r(y^s) - r(y^g)\big)\sum_{t=1}^{T} \log p_\theta\big(y_t^s \mid y_{<t}^s\big). \quad (2)$$

Because the baseline $r(y^g)$ does not depend on y^s, subtracting it leaves the expected gradient of Eq. 1 unchanged but reduces its variance (Rennie et al., 2017); samples that beat the greedy decode are reinforced and samples worse than greedy are discouraged. Greedy decoding is also the standard inference strategy in code completion, making it a natural reference behaviour.

We do not employ PPO-style optimization (Schulman et al., 2017), as it requires training an additional reward estimation model, whereas our setting already provides deterministic reward functions. Similarly, GRPO (Shao et al., 2024) requires computing gradients for multiple sampled rollouts per prompt, which significantly increases computational cost. Given these considerations, SCST offers a practical trade-off between training stability and efficiency while directly optimizing the desired reward signal.

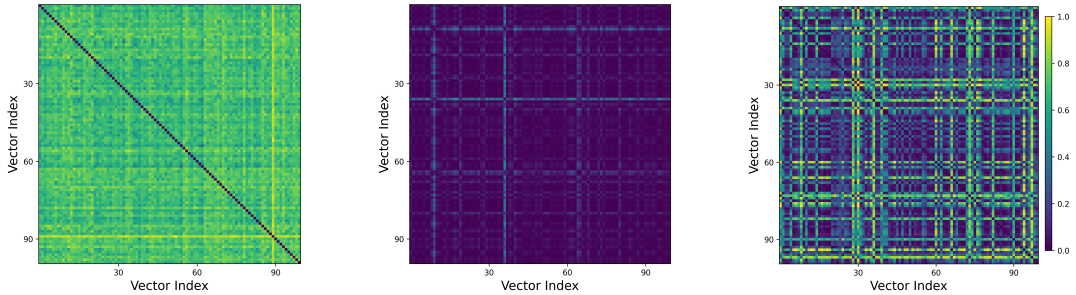
3.4 Cosine Alignment Loss

During initial experiments on training the projection from encoder representations into the LLM embedding space, we observed that the projection MLP often collapsed to an almost one-dimensional subspace: the angles between projected vectors converged to nearly zero across most pairs (Figure 2b), even though the encoder itself is expressive and produced embeddings with pairwise cosine similarities broadly distributed in $[0.0, 1.0]$ (Figure 2a).

This behavior is problematic, since our goal is to preserve the distinctions between retrieved text chunks. To address this collapse and retain the relative differences among encoder embeddings after projection, we introduce a specialized *Cosine Alignment loss*:

$$\mathcal{L}_A(\theta) = \frac{1}{B^2}\|S_C(y_{\text{enc}}) - S_C(y_{\text{proj}})\|_F^2, \quad (3)$$

where $S_C(\cdot)$ denotes the cosine similarity matrix between vectors of the output batch; y_{enc} and y_{proj} represent the encoder and projection output batches, respectively; and B is the batch size. This



(a) Encoder output (b) Collapsed projector output (c) Well-separated projector output

Figure 2: Pairwise cosine distances between vector outputs. While the encoder representations remain well-separated (a), the projected vectors may collapse, becoming nearly indistinguishable (b). Introducing the Cosine Alignment loss (Eq. 3) helps preserve the distinctions among the projections, preventing excessive overlap (c).

loss enforces preservation of pairwise cosine similarities within a batch by minimizing the mean squared error between the two similarity matrices, each of size $B \times B$. After applying it, projected vectors recover the well-separated structure of the encoder embeddings (Figure 2c).

3.5 KL Loss

In contrast to recent findings (Cheng et al., 2024), we observe that adding a KL-divergence loss between models trained with compressed versus uncompressed retrieved context does not improve prediction quality on code completion. This is notable because xRAG (Cheng et al., 2024) assigns the KL component a relatively large weight (2.0) compared to the negative log-likelihood (1.0), motivated by ablations on instruction-tuned models for QA. As shown in Table 1, reproducing the xRAG loss yields performance below the unaugmented baseline, indicating that KL distillation is not a reliable training signal for projector learning in our task.

3.6 Final Loss Function

We optimize our model using the following composite loss function:

$$\mathcal{L}(\theta) = \alpha_{CE}\mathcal{L}_{CE}(\theta) + \alpha_R\mathcal{L}_R(\theta) + \alpha_A\mathcal{L}_A(\theta), \tag{4}$$

where the coefficients α_{CE}, α_R, and α_A are weighting factors. These weights are selected through hyperparameter tuning using Optuna[3] (Akiba et al., 2019). These and other training hyperparameters for all trained models are listed in Section 4.1. The dynamics of the three loss components and their correspondence to EM and ES are shown in Figure 3 of Appendix C.1.

[3] https://optuna.org

4 Experiments

4.1 Experimental Setup

We run 1.5B-parameter ablations on The Stack (Kocetkov et al., 2022) and report our main 7B results on The Stack v2 (Lozhkov et al., 2024) (Python and Java); dataset construction and repository-level splitting are detailed in Appendix A.

Task Formulation Code completion is formulated as a fill-in-the-middle (FIM) task. Given the left context (lines preceding the target code chunk) and the right context (lines following it) from a source file, the model must generate the missing middle segment. Each target segment consists of n_t consecutive lines ($1 \leq n_t \leq 9$), with n_t sampled from a Poisson distribution to encourage diversity in completion length. This setup reflects realistic code editing scenarios while avoiding bias toward fixed-length completions.

Cross-File Context Construction. To incorporate repository-level information, we retrieve relevant context from non-target files within the same repository. All such files are segmented into overlapping chunks of $10 \times n_t$ lines, with an overlap of $5 \times n_t$ lines. From this pool, we select the top 10 most relevant chunks based on Jaccard similarity over tokenized code. We chose Jaccard from several alternatives (BM25, dense retrievers based on UniXCoder and Jina v2) after a comparison on 1,600 completion tasks; the full retrieval-strategy ablation is reported in Appendix A.3.

Evaluation Protocol During training, we use the full available target length, enabling the model to learn multi-line generation.

We report results on established repository-level

Table 1: Ablation study of projection-training approaches. Results without context augmentation are denoted w/o CFC; results with uncompressed cross-file context (w/ CFC) are highlighted in gold. α_{KL} denotes the weight of the KL loss, other loss terms follow Section 3.6. Metrics are reported on an evaluation subset prepared from The Stack ($4.3k$ samples).

Method	α_{CE}	α_R	α_A	α_{KL}	CE Loss ↓	EM ↑	ES ↑
Base Model w/o CFC	-	-	-	-	0.97	45.97	66.57
Base Model w/ CFC	-	-	-	-	0.99	50.87	69.43
LLaVA (Liu et al., 2023)	1.0	0.0	0.0	0.0	**0.80**	38.57	63.6
REINFORCE-only	0.0	1.0	0.0	0.0	5.18	40.61	63.91
CE + Cos. Align.	0.9	0.0	0.1	0.0	0.89	42.3	64.43
xRAG (Cheng et al., 2024)	1.0	0.0	0.0	2.0	0.84	44.0	65.5
CoRoVA (ours)	0.9	0.1	0.1	0.0	1.02	**47.66**	**68.74**

code-completion benchmarks (Table 2), which are single-line completion tasks: given the left and right context, the model must predict the single missing line. Since the single-line case lies within the target-length distribution used during training ($1 \leq n_t \leq 9$), this evaluation probes an in-distribution slice of the model's behavior rather than a separate regime.

Before computing EM and ES metrics, we strip leading and trailing whitespace from predictions and targets to avoid *artificial metric inflation* from trivially predictable formatting tokens (e.g., indentation). This normalization ensures that EM and ES primarily reflect code-level correctness rather than superficial whitespace accuracy.

When comparing against the LLM with non-compressed text context, the top-10 retrieved contexts are concatenated into one sequence, truncated to 512 tokens, and then concatenated with the same code completion prompt (Fig. 1a). The code completion prompt budget (without retrieved context) is 2,000 tokens for both methods. These context-length budgets follow prior work on retrieval-augmented code generation (RACG), such as Repoformer (Wu et al., 2024). As a result, the input sequence length in CoRoVA is 502 tokens shorter than in conventional RAG.

Training We train 2- and 3-layer MLP projection modules that map sentence-encoder outputs (e.g., UniXCoder or Qwen3-Embedding) into the dimension of the code LLM embedding space. For each sample, the top-10 cross-file contexts – encoded and projected into LLM representations at the time of RAG database creation – are concatenated with the code completion prompt embeddings before being passed to the LLM (Fig. 1b).

During training, only the projection weights are updated, while both the encoder and the LLM remain frozen. Optimization uses the joint loss of Section 3.6. Cross-entropy is only computed over the sequence after the `<|fim_middle|>` special token, and for the REINFORCE term we generate 50 tokens with greedy decoding and evaluate EM/ES on the resulting sequence. The full list of training hyperparameters, an analysis of single-stage vs. two-stage training, and an unsuccessful projector-pretraining attempt are deferred to Appendix B.

4.2 Main Results

Overall Benchmark Performance. Table 2 presents our main results across multiple established code completion benchmarks. We evaluate CoRoVA, our approach for retrieved-context compression, against two key baselines that isolate the effect of cross-file context usage: (i) **w/o CFC** – uses only local in-file context (no cross-file retrieval); (ii) **w/ CFC** – incorporates the retrieved cross-file context (CFC) without compression (Figure 1a).

All models share the same underlying LLM (Qwen2.5-Coder-7B) and retrieval mechanism; they differ solely in how retrieved context is integrated, enabling a clean ablation of compression efficiency.

Despite the negligible latency impact introduced by the additional 10 tokens, our approach surpasses the w/o CFC baseline on EM and ES metrics by a sizable margin, which makes CoRoVA preferable in latency-limited environments such as IDE code completion, where vanilla RAG introduces a noticeable latency impact in the range of 20–38%. Detailed latency measurements are presented in Section 4.3 (with secondary tables in Appendix D); qualitative completion examples are provided in Appendix C.7. We also evaluate the *resilience rate* of CoRoVA – the percentage of inputs for which the response remains correct after retrieval augmentation (Cheng et al., 2024) – and the execution-based Pass@5 on RepoEval Function Completion; CoRoVA is competitive with the uncompressed-context baseline on both (Figure 5 of Appendix C.4; Pass@5 in Appendix C.6). We further characterise this trade-off as a quality–cost Pareto frontier and analyse the sensitivity to the number of projected vectors k (the top-k retrieved chunks) in Appendix C.3: CoRoVA dominates naive context truncation at matched token budgets, whereas uncompressed RAG must spend roughly $50\times$ more tokens to surpass it.

Table 2: Results on code-completion benchmarks. All methods use Qwen2.5-Coder-7B as the base code-generating LLM. For Java on The Stack v2, we use the test split of the dataset.

Language	Benchmark	Model	Seq. Len.	EM ↑	ES ↑	CodeBLEU↑
Python	CCEvalLong (Wu et al., 2024)	w/o CFC	2000	45.64	71.75	55.82
		w/ CFC	2512	**49.74**	73.2	**58.01**
		CoRoVA (ours)	2010	47.16	**73.35**	57.46
	RepoEval (Zhang et al., 2023)	w/o CFC	2000	58.13	77.98	58.09
		w/ CFC	2512	**64.56**	**81.69**	**64.34**
		CoRoVA (ours)	2010	60.56	80.46	59.67
		Context Pruning	2010	57.91	78.1	58.05
		Context Summar.	2010	58.04	78.26	58.23
		CodePromptZip (He et al., 2025)	2010	57.91	78.3	58.26
	RepoEval Api (Zhang et al., 2023)	w/o CFC	2000	48.75	74.13	52.32
		w/ CFC	2512	**55.00**	**79.31**	**57.60**
		CoRoVA (ours)	2010	51.31	77.95	55.56
Java	The Stack v2 (Lozhkov et al., 2024)	w/o CFC	2000	68.3	82.8	63.96
		w/ CFC	2512	70.92	84.43	65.69
		CoRoVA (ours)	2010	**71.2**	**84.87**	**66.03**
	RepoEval-Updated (Liu et al., 2024)	w/o CFC	2000	52.71	76.49	64.13
		w/ CFC	2512	**55.63**	**78.03**	**66.01**
		CoRoVA (ours)	2010	54.11	77.21	65.27

Comparison to Compression Baselines. On RepoEval, we compare CoRoVA against token-space compression approaches that target a comparable context budget, including token pruning – compression of the context into a fixed token budget by selecting the most representative tokens; and summarization – the model is prompted to produce a maximally faithful fixed-length summary of the retrieved cross-file context (Table 2). Both pruning and summarization are performed using Qwen2.5-Coder-7B Instruct, and we additionally include the code-specific CodePromptZip (He et al., 2025). Methods operating in token space *fail to preserve completion quality at the extreme compression level* required here, whereas CoRoVA *retains the benefits of cross-file context*. We also reproduce the original xRAG pipeline on RepoEval (Appendix C.5) and find that document-oriented compression does not transfer well to code.

Objective Ablation. Table 1 presents the results of ablation studies across different loss formulations, comparing four configurations: Cross-Entropy only (LLaVA-style), REINFORCE only, Cross-Entropy with Cosine Alignment, and Cross-Entropy with KL Loss (the xRAG objective). As discussed in Section 3.2, relying solely on the Cross-Entropy objective degrades performance on both EM and ES metrics. Conversely, optimizing exclusively with the REINFORCE loss leads to uncontrolled entropy growth and fails to outperform the w/o CFC baseline. In contrast, only a carefully balanced combination of all three loss components (Section 3.6) yields consistent improvements across the target metrics; the corresponding loss-vs-metric dynamics are plotted in Figure 3 of Appendix C.1.

Architecture Ablation. We ablate the choice of encoder and projector depth, comparing UniXCoder (Guo et al., 2022) on AST representations with Qwen3-Embedding-0.6B (Zhang et al., 2025) on raw code, and 2-layer vs. 3-layer MLP projectors. Qwen3-Embedding-0.6B outperforms UniXCoder, and the 3-layer projector further improves both EM and ES at the cost of roughly $4\times$ more trainable parameters; the full table appears in Appendix C.2.

4.3 Online Inference Latency (TTFT/TPOT)

CoRoVA shifts *context understanding* off the critical inference path: the Encoder+Projector runs at repository indexing time, so at inference only the precomputed representations are appended to the prompt and user-perceived latency depends on the *effective sequence length* (indexing overhead is quantified in Appendix E).

Two deployment regimes dominate LLM serving. *Prefill-decode mixing* (e.g., vLLM (Kwon et al., 2023)) interleaves prefill and decode in one engine; it is utilization-friendly, but the two phases contend for the same resources, making it hard to independently control TTFT (*time-to-first-token*) and TPOT (*time-per-output-token*) under SLAs (Agrawal et al., 2024). *Disaggregated prefill-decode* (e.g., DistServe (Zhong et al.,

Table 3: Effect of context compression on TTFT on an NVIDIA A100, comparing disaggregated prefill-decode serving (transformers) against prefill-decode mixing (vLLM). Under disaggregation, context compression directly leads to a similar decrease in TTFT, enabling an earlier start of response generation. Under prefill-decode mixing, gains are smaller because decode-side workload dominates end-to-end latency.

Seq. compression	Model	TFLOPs	TTFT (Transformers)	TTFT (vLLM)
2512 → 2010 ↓ 20%	Qwen2.5-Coder-1.5B	30.9 → 24.8 ↓ 20%	198.2 → 156.6 ↓ 21%	74.7 → 68.2 ↓ 9%
	Qwen2.5-Coder-7B	141.5 → 113.8 ↓ 20%	668.6 → 541.1 ↓ 19%	198.3 → 166.5 ↓ 16%
	Qwen2.5-Coder-14B	279.9 → 225.1 ↓ 20%	822.8 → 661.3 ↓ 20%	349.8 → 291.7 ↓ 17%
2000 → 1510 ↓ 25%	Qwen2.5-Coder-1.5B	24.7 → 18.7 ↓ 24%	157.4 → 113.4 ↓ 28%	65.3 → 58.4 ↓ 11%
	Qwen2.5-Coder-7B	113.2 → 85.5 ↓ 24%	540.0 → 406.8 ↓ 25%	179.0 → 134.0 ↓ 25%
	Qwen2.5-Coder-14B	224.0 → 169.1 ↓ 25%	662.2 → 496.3 ↓ 25%	291.5 → 232.9 ↓ 20%
1500 → 1010 ↓ 33%	Qwen2.5-Coder-1.5B	18.5 → 12.5 ↓ 32%	112.2 → 69.7 ↓ 38%	58.9 → 50.3 ↓ 15%
	Qwen2.5-Coder-7B	84.9 → 57.2 ↓ 33%	406.4 → 282.2 ↓ 31%	138.0 → 112.4 ↓ 19%
	Qwen2.5-Coder-14B	168.0 → 113.2 ↓ 33%	495.6 → 339.6 ↓ 31%	238.2 → 174.6 ↓ 27%

2024)) places the two phases on separate pools, eliminating interference and decoupling the two metrics. In interactive IDE settings TTFT dominates the user experience – tokens must start appearing quickly so the developer can begin reviewing the suggestion.

Table 3 reports the headline results on an NVIDIA A100; TFLOPs are listed alongside TTFT to expose the system-independent prefill cost (prefill-only and TPOT measurements are in Appendix D). Under disaggregation, shorter contexts translate directly into lower prefill latency and proportionally smaller TTFT; under colocated prefill-decode the gains are smaller because decode-side workload dominates end-to-end latency.

5 Conclusion

Retrieval-augmented code completion can substantially improve prediction quality, but the additional retrieved tokens inflate the prefill cost and degrade TTFT, undermining usability in interactive IDE settings. We introduced CoRoVA, a vector-augmented code-completion pipeline that compresses retrieved cross-file context into a small set of continuous vectors via embedding projection, improving completion quality *over the no-context baseline* while preserving low-latency responsiveness and yielding a 20–38% reduction in TTFT compared to vanilla RAG. Rather than seeking parity with uncompressed RAG, CoRoVA targets the quality–latency Pareto frontier, recovering a substantial fraction of RAG's quality gain at roughly 50× fewer context tokens and outperforming naive truncation and prior context-compression baselines at a matched token budget (Appendix C.3).

To the best of our knowledge, CoRoVA is the first LLaVA-style approach to apply compression to code generation, to explore the addition of structurally rich code modalities, to utilize base models rather than instruction-tuned ones, and to apply reinforcement learning to train the projection for downstream code-completion tasks. CoRoVA trains only a lightweight projector module while keeping the encoder and the code LLM frozen, using a composite objective that combines cross-entropy with a REINFORCE-style term that directly targets ES and EM and a cosine-alignment term that prevents representational collapse after projection. We further showed that previously proposed projector-training objectives *fail on code completion* and designed a composite loss that *consistently improves the target metrics*.

Future work could investigate more compute-intensive RL optimization strategies such as GRPO (Shao et al., 2024) to further improve EM/ES alignment, and revisit structural code signals as stronger graph-based encoders become available.

Limitations

Our study has several limitations. First, our experiments cover Python and Java; extending the evaluation to other widely used languages such as C#, C++, or JavaScript would give a broader picture of the method's generality. Second, our reinforcement-learning component relies on SCST, chosen for its training stability and efficiency; more compute-intensive optimizers such as PPO or GRPO might further improve alignment with the EM/ES metrics but were not explored here. Finally, the structural code modalities we consider are limited to ASTs encoded by

UniXCoder, as stronger publicly available graph encoders for code (e.g. for CFGs/DFGs) are currently lacking; revisiting this direction as such encoders mature is an avenue for future work.

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. 2024. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *Preprint*, arXiv:2305.13245.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A next-generation hyperparameter optimization framework. *Preprint*, arXiv:1907.10902.

Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katie Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, and 8 others. 2022. Flamingo: a visual language model for few-shot learning. *Preprint*, arXiv:2204.14198.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *Preprint*, arXiv:1607.06450.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 11711179, Cambridge, MA, USA. MIT Press.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Xin Cheng, Xun Wang, Xingxing Zhang, Tao Ge, Si-Qing Chen, Furu Wei, Huishuai Zhang, and Dongyan Zhao. 2024. xrag: Extreme context compression for retrieval-augmented generation with one token. *Preprint*, arXiv:2405.13792.

Michael Günther, Jackmin Ong, Isabelle Mohr, Alaeddine Abdessalem, Tanguy Abel, Mohammad Kalim Akram, Susana Guzman, Georgios Mastrapas, Saba Sturua, Bo Wang, Maximilian Werk, Nan Wang, and Han Xiao. 2023. Jina embeddings 2: 8192-token general-purpose text embeddings for long documents. *Preprint*, arXiv:2310.19923.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *Preprint*, arXiv:2203.03850.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. *Preprint*, arXiv:2009.08366.

Pengfei He, Shaowei Wang, and Tse-Hsun Chen. 2025. Codepromptzip: Code-specific prompt compression for retrieval-augmented generation in coding tasks with lms. *Preprint*, arXiv:2502.14925.

D Hendrycks. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

Andrew Jaegle, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. 2021. Perceiver: General perception with iterative attention. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 4651–4664. PMLR.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code. *Preprint*, arXiv:2211.15533.

Yuri Kuratov, Mikhail Arkhipov, Aydar Bulatov, and Mikhail Burtsev. 2025. Cramming 1568 tokens into a single vector and back again: Exploring the limits of embedding space capacity. *Preprint*, arXiv:2502.13063.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2021. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Preprint*, arXiv:2005.11401.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, and 48 others. 2023. Starcoder: may the source be with you! *Preprint*, arXiv:2305.06161.

Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual instruction tuning. *Preprint*, arXiv:2304.08485.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *Preprint*, arXiv:2406.07003.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently scaling transformer inference. *Preprint*, arXiv:2211.05102.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2015. Sequence level training with recurrent neural networks. *CoRR*, abs/1511.06732.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Steven J. Rennie, Etienne Marcheret, Youssef Mroueh, Jarret Ross, and Vaibhava Goel. 2017. Self-critical sequence training for image captioning. *Preprint*, arXiv:1612.00563.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *Preprint*, arXiv:2405.07530.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(34):229256.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 53270–53290. PMLR.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Tatiana Zemskova and Dmitry Yudin. 2025. 3dgraphllm: Combining semantic graphs and large language models for 3d scene understanding. *Preprint*, arXiv:2412.18450.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. 2025. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *Preprint*, arXiv:2506.05176.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. 2024. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210.

Table 4: Statistics of the filtered The Stack v2 dataset.

Language	Total Repositories	Train Examples	Val. Examples	Test Examples
Python	118,562	2,839,479	157,969	160,836
Java	226,447	5,273,006	293,694	294,715

A Datasets and Retrieval

A.1 Datasets and Splitting

For initial ablation studies and experiments with 1.5B-parameter models, we use the Python subset of The Stack (Kocetkov et al., 2022). Following prior work (Wu et al., 2024), we organize files at the repository level and apply quality filters to ensure meaningful cross-file context. Specifically, we remove repositories with fewer than 50 GitHub stars, fewer than five files, or files containing fewer than three import statements. After filtering, this dataset yields 150k code completion examples.

For all main results with the 7B-parameter models, we adopt The Stack v2 (Lozhkov et al., 2024), restricted to Python and Java repositories collected prior to September 14, 2023. To ensure sufficiently rich repository context, we apply stricter project-level constraints: each repository must contain at least ten files, and each file must have a minimum of forty lines of code, restricted to the main branch. To prevent data leakage, repositories are split into training, validation, and test sets at the repository level.

A.2 Dataset Statistics

Statistics of the filtered The Stack v2 dataset used in our main experiments are reported in Table 4. Repositories are split repository-wise to prevent data leakage; a uniformly subsampled set of 4,000 validation examples is used for fast inference-time analysis and 4,000 test examples for final metric reporting on the Java subset.

A.3 Retrieval Strategy Selection

We evaluated several retrieval strategies for selecting cross-file context, including sparse methods (BM25 and Jaccard) and dense retrievers based on cosine similarity between embeddings from UniXCoder (Guo et al., 2022) and Jina v2 (Günther et al., 2023). Each retriever-augmented model was compared against a baseline without cross-file context on a subset of 1,600 completion tasks from The Stack. Results are reported in Table 5. Overall, Jaccard similarity and UniXCoder embeddings achieve the strongest performance. Given its simplicity and lower inference latency, we adopt Jaccard similarity as the default retrieval method in all experiments.

Table 5: Comparison of different retrieval strategies.

Metric	W/o CFC	BM25	Jaccard	UniXCoder (Guo et al., 2022)	Jina v2 (Günther et al., 2023)
EM	50.50	55.56	**56.19**	56.00	54.31
ES	73.12	76.5	76.68	**76.84**	75.56

B Training Details

B.1 Hyperparameters

For our primary evaluations we use the Qwen2.5-Coder family of models, with Qwen3-Embedding-0.6B serving as the encoder. A three-layer MLP is employed as the projector, mapping the encoder dimension to twice the LLM embedding size and finally down to the LLM embedding size; a GELU activation and a LayerNorm are applied between consecutive layers. Training hyperparameters for the different model sizes are listed in Table 6. Projectors for 7B model were trained on a single node with 4× NVIDIA H100 (80 GB) GPUs (DDP, bf16). We trained for approximately eight hours and processesed roughly 2% of the filtered The Stack v2 training set.

Table 6: Hyperparameter settings used for projector training.

Hyperparameter	1.5B models	7B models
Optimizer	AdamW	AdamW
α_A (Cosine Alignment)	0.1	0.2
α_{CE} (Cross-Entropy)	0.9	0.9
α_R (REINFORCE)	0.1	0.05
Learning rate (LR)	1e-3	5e-5
LR scheduler type	cosine	cosine
Warmup ratio	0.03	0.04
Weight decay	0.0	0.0
#Epochs	3	0.01
Effective batch size	66	64
#Train samples	150k	2.8M

B.2 Single-stage vs. Two-stage Training

Whereas most prior work adopts two-stage training, CoRoVA uses a single-stage pipeline based on the composite loss function of Section 3.6. For completeness, we also evaluated a conventional two-stage pretrain-finetune pipeline for projection training.

In prior work, pretraining often relies on parallel datasets, such as paraphrase pairs in xRAG or image–caption pairs in LLaVA. Inspired by xRAG, we experimented with a similar pretraining approach, attempting to reconstruct retrieved context chunks from projected vectors by optimizing cross-entropy loss. This did not yield improve-

ments in the second stage of training, likely due to the entropy issues discussed in Section 3.2.

B.3 Pretraining the Projector with Memory Tokens

The work of Kuratov et al. (2025) demonstrates that up to 1,568 tokens can be compressed into a single continuous "memory" token by treating the token as a trainable parameter and optimizing it via backpropagation with a cross-entropy reconstruction loss. Because these continuous tokens reconstruct to reference texts, we treat them as ground truth for training our projection layer. We encode text with our encoder, project the resulting embeddings into a single token, and optimize a mixture of mean squared error (MSE) and cosine-similarity (CS) losses between the projected embedding and the trained ground-truth compressed token.

However, the space spanned by the memory tokens proved to be highly non-smooth. Identical text inputs could be compressed into vectors that are widely separated, and introducing even small perturbations to a learned memory token often results in the reconstruction of completely different text. This leads to poor generalization for an MLP module attempting to map into this space. Consequently, learning a projection into such a space requires extreme overparameterization, effectively amounting to memorizing the entire dataset. As a result, we could only overfit on a small subset of memory tokens and were unable to learn a meaningful translation into the memory token space. We leave more sophisticated pretraining of the projection module for code compression to future work.

C Additional Experimental Results

C.1 Loss Dynamics

Figure 3 shows the joint dynamics of the three loss components (Cross-Entropy, REINFORCE, Cosine Alignment) and the evaluation metrics Exact Match (EM) and Edit Similarity (ES) during training of the CoRoVA projector. The CE term decreases throughout training while EM and ES improve only when all three components are active, consistent with the objective ablation in Table 1.

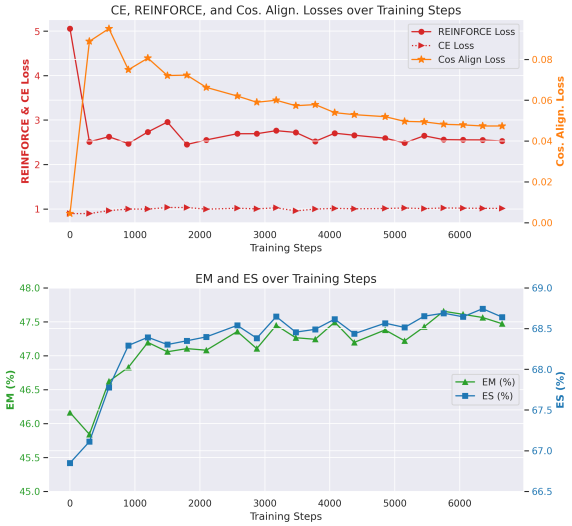


Figure 3: Relationship between the three loss components (Cross-Entropy, REINFORCE, Cosine Alignment) and the evaluation metrics Exact Match (EM) and Edit Similarity (ES).

C.2 Architecture Ablation

Table 7 reports the full ablation over encoder choice (UniXCoder on ASTs vs. Qwen3-Embedding-0.6B on raw code) and projector depth (2-layer vs. 3-layer MLP), including trainable parameter counts. Parameter differences arise from the encoder embedding dimensionality and the MLP depth. All configurations are trained for approximately 6,600 steps (3 epochs).

Table 7: Encoder and projector configurations: trainable parameter counts and downstream performance.

Encoder	Modality	Projection	#Trainable Params.	EM	ES
UniXCoder	AST	2-layer MLP	3.5M	46.69	67.65
UniXCoder	AST	3-layer MLP	16.5M	46.94	68.26
Qwen3-Embedding	Code	2-layer MLP	3.9M	47.01	68.15
Qwen3-Embedding	Code	3-layer MLP	17.3M	**47.66**	**68.74**

C.3 Number of Projected Vectors and the Quality–Cost Frontier

This section reports the sensitivity of CoRoVA to the number of projected vectors k (the "top-k" contextual tokens), clarifies how these vectors are distributed across retrieved chunks and where they are placed, and positions CoRoVA on the quality–cost frontier against uncompressed RAG.

Design: one vector per chunk, prepended. CoRoVA encodes *each* retrieved chunk into a single pooled vector and projects it to exactly one reader token (Section 3). The number of projected vectors therefore *equals* the number of retrieved chunks: a "top-k" configuration injects k reader tokens, one per chunk, and we do not allocate sev-

eral vectors to a single chunk. All projected vectors are prepended to the completion prompt and occupy ordinary, consecutive sequence positions; this placement is used throughout the paper. Consequently k is bounded by the number of retrieved chunks (ten in our setup), and CoRoVA adds k tokens against the hundreds consumed by uncompressed RAG.

Sensitivity to k and matched-budget comparison. Figure 4 sweeps $k \in \{1, 2, 3, 4, 5, 10\}$ for CoRoVA on StarCoder2-7B (RepoEval-Python, BM25 retrieval) and places every configuration on a common cost axis – the number of cross-file tokens added to the reader prompt. For uncompressed RAG we additionally vary the textual budget ($5/10/20/50$ and 512 tokens), so the two methods can be compared at *matched* token cost (Table 8). Two observations follow. First, CoRoVA quality rises with k but saturates quickly: top-5 and top-10 differ by ≤ 0.8 EM, since the retriever returns at most ten chunks and the marginal chunk adds little. Second, in the token-scarce regime CoRoVA is Pareto-optimal: with 5–10 injected tokens it reaches EM 52.5/ES 75.5, whereas *truncating* uncompressed RAG to the same 5–50 tokens stays at or below the no-context baseline (EM 50.6–51.3). Uncompressed RAG surpasses CoRoVA only once granted ∼300–512 tokens (EM 57.7–59.2), i.e. roughly $50\times$ more context tokens.

Takeaway and expectation for larger k. CoRoVA does not aim to *match* full RAG quality; it occupies the low-latency end of the quality–cost frontier, recovering a substantial share of RAG's gain at a small fraction of its token cost and dominating naive truncation at equal cost. This is consistent with the main results (Table 2), where CoRoVA improves over the no-context baseline on every benchmark while adding only ten tokens and, at a matched budget, surpasses learned token-space compression baselines – context pruning, summarization, and CodePromptZip (He et al., 2025). Because gains already saturate by $k = 5$ and the retriever returns ten chunks, we expect top-20 to yield only marginal additional improvements under the current one-vector-per-chunk design; the limiting factor is per-chunk compression fidelity rather than the number of chunks. Allocating multiple vectors per chunk and ablating vector placement are natural extensions we leave to future work.

Table 8: Matched-budget operating points behind Figure 4 (StarCoder2-7B, RepoEval-Python, BM25). "Budget" is the number of cross-file tokens added to the reader prompt; CoRoVA is capped at the number of retrieved chunks ($k \leq 10$). Bold marks the better method at equal budget.

Budget (tok)	Method	EM	ES
0	No CFC	51.12	73.94
5	RAG (truncated)	51.00	73.73
5	CoRoVA (k=5)	**52.50**	**75.36**
10	RAG (truncated)	51.31	73.93
10	CoRoVA (k=10)	**52.50**	**75.46**
20	RAG (truncated)	50.94	73.74
50	RAG (truncated)	50.56	73.38
311	RAG (1 fragment)	57.69	78.11
512	RAG (5–10 frag.)	59.13	78.49

C.4 Robustness to Retrieval Augmentation

Figure 5 shows the resilience rate – the percentage of instances where the model's response remains correct both before and after retrieval augmentation (Cheng et al., 2024) – for CoRoVA compared to a model using uncompressed cross-file context. Overall, the rates are comparable, with CoRoVA showing a slight advantage on average.

C.5 Comparison to the Original xRAG Pipeline

Table 9 reports results for the original xRAG pipeline[4] on RepoEval. We do not include xRAG in the main comparison (Table 2) because it relies on a general-purpose Mistral-7B model and a compression mechanism trained on retrieved natural-language documents for downstream QA. The results suggest that document-oriented compression objectives do not translate well to code completion, yielding at best marginal gains in CodeBLEU (Ren et al., 2020) while degrading other metrics such as Edit Similarity (Zhang et al., 2023).

Table 9: Performance of the original xRAG pipeline on RepoEval.

Model	Seq. Length	EM ↑	ES ↑	CodeBLEU↑
w/ CFC	2000	0.19	14.05	3.88
xRAG	2010	0.19	13.42	4.10

C.6 Execution-based Evaluation (Pass@k)

To rigorously evaluate the functional correctness of generated code beyond syntactic validity, we adopt the execution-based Pass@k metric (Chen et al., 2021). We select the RepoEval Function

[4] https://huggingface.co/Hannibal046/xrag-7b

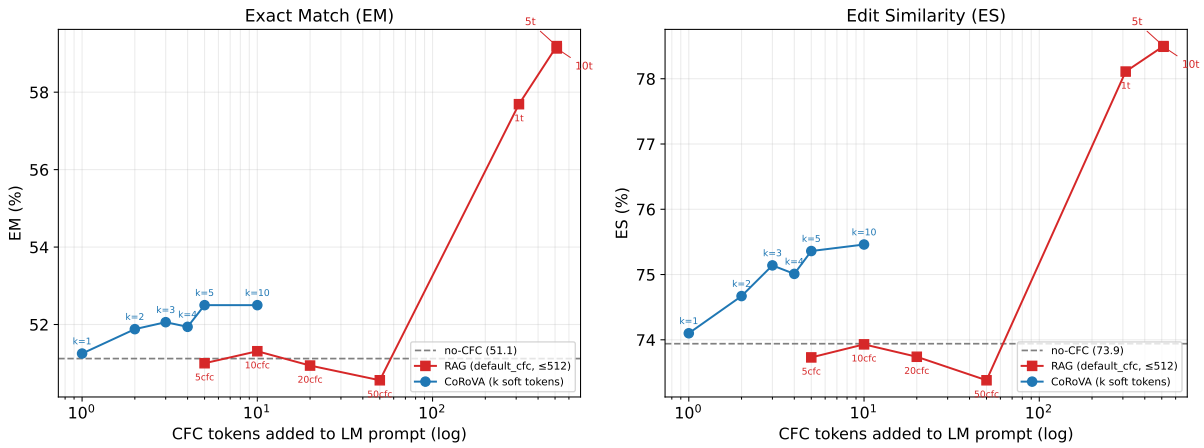


Figure 4: Quality–cost frontier on StarCoder2-7B (RepoEval-Python, BM25 retrieval). **Left:** Exact Match; **right:** Edit Similarity, against the number of cross-file tokens added to the reader prompt (log scale). CoRoVA (blue) injects k projected vectors ($k \in \{1, \ldots, 5, 10\}$, one per retrieved chunk); uncompressed RAG (red) is shown at textual budgets of $5/10/20/50$ and 512 tokens (`1t`/`5t`/`10t` denote the number of fragments under the 512-token budget, which saturates so the points overlap). The dashed line is the no-context baseline. At matched budgets (≤ 50 tokens) CoRoVA dominates; uncompressed RAG overtakes it only with $\sim 50\times$ more tokens.

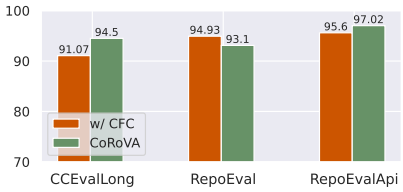


Figure 5: Resilience rate of two augmentation methods – uncompressed retrieved cross-file context (w/ CFC) and CoRoVA – relative to a Qwen2.5-Coder-7B baseline without RAG.

Completion benchmark (Zhang et al., 2023) because each function has manually verified test suites with strict coverage requirements, unlike RepoEval's line/API-completion subsets where tests frequently lack coverage guarantees for the completion targets.

While open-source Pass@k evaluation frameworks exist[5], we encountered irreproducible test environments and dependency conflicts when attempting to reuse them. To ensure evaluation integrity, we implemented our own pipeline. From the benchmark's 8 repositories, only `leopard/ai-betty`, `deepmind/tracr`, and `amazon-science/patchcore-inspection` passed all unit tests using their original, unmodified code; we exclude the other five repositories because their test suites are broken. This yields a benchmark of 244 functions. Table 10 reports Pass@5 results. CoRoVA achieves the best score on `patchcore-inspection` (+3.1% over the strongest baseline) and `ai-betty` (+1.6%), demonstrating effective context utilization. On `tracr`, all context-aware methods underperform the no-context baseline, suggesting repository-specific sensitivity to cross-file dependencies, but CoRoVA still outperforms the w/ CFC baseline.

[5] https://github.com/Elendil3703/RepoEval_Exec

Table 10: Pass@5 (%) on the RepoEval Function Completion benchmark, reported per repository. Qwen2.5-Coder-7B is the base LLM. Repositories: `amazon-science/patchcore-inspection`, `deepmind/tracr`, `leopard/ai-betty`.

Model	Seq. Length	patchcore-inspection	tracr	ai-betty
w/o CFC	2000	64.92	**56.11**	63.14
w/ CFC	2512	63.09	52.39	63.97
CoRoVA (ours)	2010	**66.94**	52.76	**64.99**

C.7 Qualitative Completion Examples

Table 11 presents qualitative examples illustrating how CoRoVA handles code completion tasks from the benchmarks of Table 2, compared with both the baseline model without cross-file context (w/o CFC) and the model using uncompressed cross-file context (w/ CFC). Although some benchmark tasks involve multi-line completions, only one-line cases are displayed to fit more examples.

D Detailed Latency Measurements

This section expands on the headline latency results presented in Section 4.3. Table 12 reports latency reduction in a prefill-only regime (1-token generation) for both the `transformers` and vLLM frameworks. Table 13 reports TPOT measurements on an NVIDIA A100 for the same two

Table 11: Examples of generated code for the benchmarks of Table 2.

Method	Code Completion
ground truth	`pos += slice_length`
w/o CFC	`return`
w/ CFC	`yield ""`
CoRoVA	`pos += slice_length`
ground truth	`fields_values[name] = NAO`
w/o CFC	`if name in fields:`
w/ CFC	`if name in fields:`
CoRoVA	`fields_values[name] = NAO`
ground truth	`trial_metadata: Iterable[UnitMetadataUpdate],`
w/o CFC	`trial_metadata: Iterable[`
w/ CFC	`trial_metadata: Iterable[UnitMetadataUpdate],`
CoRoVA	`trial_metadata: Iterable[key_value_pb2.KeyValue],`
ground truth	`from jax import random`
w/o CFC	`import random`
w/ CFC	`from jax import random`
CoRoVA	`from jax import random`
ground truth	`from fortuna.prob_model.posterior.map.map_stat`
w/o CFC	`cfg = compile_config(cfg, create_cfg=create_cfg)`
w/ CFC	`from fortuna.prob_model.posterior.map.map_ste`
CoRoVA	`# Get env_fn from env_setting.`

frameworks (`transformers` in the disaggregated prefill-decode mode, vLLM in the prefill-decode mixing mode).

Table 12: Latency reduction in prefill-only regime (generation of 1 token) for the transformers and vLLM frameworks.

Seq. compression	Model	TTFT (Transformers)	TTFT (vLLM)
2512→2010 ↓ 20%	Qwen2.5-Coder-1.5B	198.2 → 159.3 ↓ 20%	79.1 → 67.0 ↓ 15%
2512→2010 ↓ 20%	Qwen2.5-Coder-7B	668.1 → 539.1 ↓ 19%	197.4 → 165.9 ↓ 16%
2512→2010 ↓ 20%	Qwen2.5-Coder-14B	820.9 → 661.1 ↓ 19%	351.5 → 291.2 ↓ 17%
2000→1510 ↓ 24%	Qwen2.5-Coder-1.5B	159.9 → 121.2 ↓ 24%	65.9 → 56.4 ↓ 14%
2000→1510 ↓ 24%	Qwen2.5-Coder-7B	539.1 → 406.3 ↓ 25%	164.4 → 135.8 ↓ 17%
2000→1510 ↓ 24%	Qwen2.5-Coder-14B	660.6 → 495.8 ↓ 25%	290.0 → 240.9 ↓ 17%
1500→1010 ↓ 33%	Qwen2.5-Coder-1.5B	120.7 → 75.5 ↓ 37%	56.4 → 48.7 ↓ 14%
1500→1010 ↓ 33%	Qwen2.5-Coder-7B	405.4 → 281.0 ↓ 31%	136.5 → 104.6 ↓ 23%
1500→1010 ↓ 33%	Qwen2.5-Coder-14B	494.9 → 339.6 ↓ 31%	240.2 → 191.4 ↓ 20%

Table 13: Impact of context compression on TPOT on an NVIDIA A100 for transformers (disaggregated prefill-decode) and vLLM (prefill-decode mixing).

Seq. compression	Model	TPOT (Transformers)	TPOT (vLLM)
2512→2010 ↓ 20%	Qwen2.5-Coder-1.5B	23.6 → 23.2 ↓ 2%	5.3 → 5.3 ↓ 0%
	Qwen2.5-Coder-7B	27.2 → 25.4 ↓ 7%	11.7 → 11.7 ↓ 0%
	Qwen2.5-Coder-14B	58.5 → 52.8 ↓ 10%	22.3 → 21.8 ↓ 2%
2000→1510 ↓ 24%	Qwen2.5-Coder-1.5B	23.5 → 23.1 ↓ 2%	5.3 → 5.6 ↑ 5%
	Qwen2.5-Coder-7B	25.1 → 24.1 ↓ 4%	11.7 → 11.6 ↓ 1%
	Qwen2.5-Coder-14B	52.7 → 47.0 ↓ 11%	21.8 → 21.8 ↓ 0%
1500→1010 ↓ 33%	Qwen2.5-Coder-1.5B	23.2 → 23.5 ↑ 1%	5.4 → 6.3 ↑ 16%
	Qwen2.5-Coder-7B	24.1 → 24.1 ↓ 0%	11.6 → 11.5 ↓ 1%
	Qwen2.5-Coder-14B	46.8 → 41.1 ↓ 12%	21.7 → 21.4 ↓ 1%

E Offline Indexing Cost

CoRoVA precomputes compressed context representations during indexing, so its offline cost depends on how quickly we can (i) chunk a repository and (ii) run the Encoder + Projector over the resulting chunks. To quantify this overhead, we sample GitHub repositories from The Stack dataset across multiple repository-size quantiles and measure end-to-end time to build the retrieval database: chunking the codebase, encoding each chunk, and applying the projector to obtain the stored compressed representations.

Unless stated otherwise, chunking is performed using a single CPU process, while encoding and projection run on a single A100 GPU. We do not implement parallelism across these stages, making the reported numbers conservative upper bounds. In practice, indexing can be parallelized across files and performed incrementally as repositories change.

Results are shown in Table 14. Switching between a 2-layer and 3-layer projector has a negligible impact on database build time, so we report the 3-layer projector configuration throughout. In contrast, encoder choice substantially affects indexing throughput: the UniXCoder-based pipeline can be up to 10× faster than Qwen3-Embedding in our setup. Overall, CoRoVA's offline indexing overhead is practical for IDE workflows and can be further reduced with standard parallel and incremental indexing strategies.

Table 14: Time to build the RAG database (chunking, computing encoder output, and projection). Qwen+3L = Qwen3-Embedding encoder + 3-layer projector (with 2× expansion); UnxC+3L = UniXCoder encoder + 3-layer projector.

Repository	Lines	Files	Chunks	Chunking	Qwen+3L	UnxC+3L
numpy/numpy	117,537	266	9,953	4s	30s	3s
apache/airflow	245,463	896	21,678	9s	54s	7s
Azure/azure-sdk-for-python	1,077,334	2,107	94,078	45s	4m 16s	39s
hawaii-clean-energy-metrics/hcem	1,083,308	51	98,858	26s	3m 40s	24s
ComputationalReflection/stypy	2,955,552	1,028	240,714	2m 28s	15m 46s	2m 22s