From Fragments to Functionality: Repository-Level Code Generation with Software Modularity

XIANGYU ZHANG and YU ZHOU*, Nanjing University of Aeronautics and Astronautics, China
GUANG YANG, Zhejiang University, China
WEI CHENG and WENHUA YANG, Nanjing University of Aeronautics and Astronautics, China
DAVID LO, Singapore Management University, Singapore
TAOLUE CHEN†, Birkbeck, University of London, UK

Repository-level code generation refers to synthesizing code within the context of the entire software project, requiring, among others, modeling of complex interdependencies across multiple files. However, traditional code models with standard attention mechanisms have difficulty scaling to longer code, limiting their ability to capture long-range dependencies at the repository level. We present COREGEN, a novel attention mechanism under the Transformer architecture for repository context understanding, guided by the core principle of high cohesion and low coupling in software modularity. COREGEN utilizes a hierarchical architecture that separates local semantic encoding from global dependency modeling. At the local level, it performs line-level semantic extraction and representation; at the global level, it constructs dependency-driven cohesion and coupling masks to guide attention towards intra-module information and critical inter-module context, allowing the model to maintain internal consistency within modules while attending to relevant external components selectively. Extensive experiments demonstrate COREGEN's superior performance in repository-level code modeling and generation across various benchmarks.

CCS Concepts: • **Software and its engineering → Automatic programming**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Repository-level code generation, Attention mechanism, Transformers, Large language models

ACM Reference Format:
Xiangyu Zhang, Yu Zhou, Guang Yang, Wei Cheng, Wenhua Yang, David Lo, and Taolue Chen. 2026. From Fragments to Functionality: Repository-Level Code Generation with Software Modularity. 0, 0, Article 0 (2026), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The advent of large language models (LLMs) has revolutionized software development through their sophisticated code generation capabilities [5, 20, 30, 49]. While early research focused primarily

*Corresponding author.
†Corresponding author.

Authors' addresses: Xiangyu Zhang, zhangx1angyu@nuaa.edu.cn; Yu Zhou, zhouyu@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; Guang Yang, novelyg@outlook.com, Zhejiang University, Hangzhou, China; Wei Cheng, chengweii@nuaa.edu.cn; Wenhua Yang, ywh@nuaa.edu.cn, Nanjing University of Aeronautics and Astronautics, Nanjing, China; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore; Taolue Chen, t.chen@bbk.ac.uk, Birkbeck, University of London, London, UK.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
XXXX-XXXX/2026/0-ART0 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

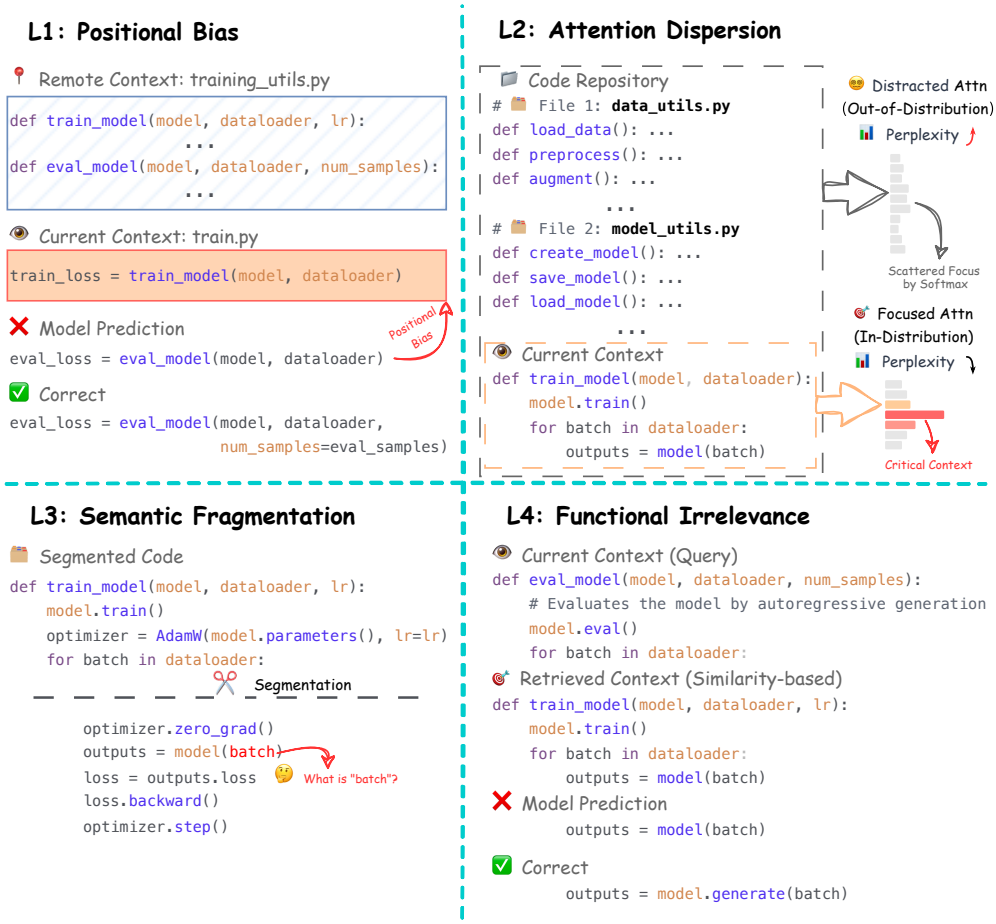


Fig. 1. Limitations of existing methods. L1 and L2 show limitations with long context generation methods for repository-level code generation; L3 and L4 show limitations with retrieval-augmented methods.

on generating isolated code snippets (mostly at the function/method level), recent advances have shifted toward the more ambitious goal of repository-level code generation [32, 56, 64, 65]. This paradigm shift entails generating code within the broader context of entire repository, requiring models to comprehend complex program structures from individual functions to inter-module dependencies, enabling the generation of code that is both functionally correct and contextually appropriate within large-scale software ecosystems [32, 65]. These enhanced capabilities, when integrated into integrated development environments (IDEs), can significantly boost developer productivity [62].

The fundamental challenge of repository-level code generation lies in effectively handling extensive and heterogeneous context of software repositories while identifying critical dependencies to ensure that the generated code aligns with the broader repository semantics [20]. However, language models are generally limited by the ability of their attention mechanisms to handle input sequences that are much longer than those seen during training [26, 29, 35]. This deficiency leads to a restricted context window that may severely limit the model's ability to capture distant, yet semantically crucial, dependencies within large-scale repositories [34, 45].

To address the challenge, current approaches directly adopted techniques from natural language processing (NLP), primarily falling into two primary paradigms: long context generation (LCG) and retrieval-augmented generation (RAG). In a nutshell, the former handles long sequences using constrained attention, while the latter retrieves relevant code fragments to avoid processing the entire repository [31, 46, 53, 58, 64, 67]. Despite their success in NLP applications, these approaches exhibit limitations when applied in software engineering, especially repository-level code generation.

LCG methods are typically designed to efficiently handle long sequences by introducing constraints that mitigate the limited length generalizability of attention mechanisms. While effective for improving scalability, these constraints introduce two challenges in large-scale code understanding:

- **L1: Positional Bias.** LCG methods commonly restrict attention to nearby tokens to prevent out-of-distribution behavior when the sequence length exceeds training horizon [7, 46, 60]. This restriction, however, limits the model's capacity to capture long-range dependencies critical in code understanding, where semantically related components (declarations, implementations, references) are often positioned far apart [67, 68].
- **L2: Attention Dispersion.** To alleviate positional bias, some LCG variants [67] adopt context-aware mechanisms that promote a more even distribution of attention across the entire code repository. While this may improve contextual awareness, the widespread dispersion of attention can dilute the model's focus on salient information, leading to increased perplexity and degraded performance [27, 69].

As illustrated in the top-left part of Figure 1, LCG methods often prioritize local context over distant dependencies, leading to errors such as generating function calls without properly including parameters from further out in the context. The top-right part shows that attempts to extend attention across long sequences in repository-scale scenarios lead to scattered focus, raising perplexity and negatively impacting the quality of the generated code.

In contrast, RAG methods bypass the need of processing entire codebases by retrieving and integrating semantically similar code fragments during inference [12, 13, 25]. However, they face distinct challenges when applied to code repositories:

- **L3: Semantic Fragmentation.** In RAG systems, codebases are typically partitioned into retrievable fragments using naive strategies, such as random or syntax-based partitioning, where each fragment is treated as an independent and self-contained unit. Such partitioning may break the logical structure and semantic coherence across code units, potentially reducing retrieval precision and the usefulness of the retrieved context [58].
- **L4: Functional Irrelevance.** Current retrieval mechanisms rely on embedding similarity, which often favors code fragments that are lexically or structurally similar to the query but not functionally dependent on the target code (e.g., neglecting method invocations, variable scoping and inheritance hierarchies) [64].

As illustrated in the bottom-left part of Figure 1, indiscriminate code segmentation can disrupt continuity, forcing the model to interpret variables or methods that are undefined within the segmented context, thereby reducing retrieval accuracy. Finally, the bottom-right part presents an example where over-reliance on semantically similar but functionally irrelevant fragments leads to context-inconsistent code generation.

To address these limitations, we revisit repository-level code generation through the lens of software modularity [42]. Fundamental software engineering principle stipulates high cohesion within modules and low coupling across modules [52], which suggests that effective repository-level code generation methods should prioritize tightly connected intra-module content while selectively integrating sparse—yet essential—inter-module dependencies to attend [23, 40], particularly in

light of the limited length generalization capabilities of the attention mechanism. Existing NLP approaches are misaligned with these principles: LCG methods preserve cohesion through constrained attention but often fail to capture essential coupling; RAG methods improve scalability through retrieval but may fragment cohesive units and overlook deeper functional dependencies.

Our contribution. We propose COREGEN (COhesion-coupling aware REpository-level code GENeration), a novel attention mechanism in the Transformer architecture designed for repository-level code generation. COREGEN introduces a hierarchical Transformer architecture which comprises two primary modules: local semantic extraction and global dependency modeling.

- **Local Semantics Extraction:** COREGEN extracts semantic representations at the *code-line level*, leveraging the observation that individual code lines serve as natural semantic units. This line-level extraction provides a structured foundation for subsequent global dependency modeling.
- **Global Dependency Modeling:** Artifact dependencies are intricate and widespread throughout the code repository. To effectively capture the heterogeneous relationship, we propose a (multi-level) dictionary learning strategy to encode dependencies from the line-level semantics at varying granularities. By analyzing these semantics, dictionary learning uncovers recurring patterns and encodes them as representative atoms, enabling the model to capture both explicit and implicit dependencies.

  Based on these dependency features, we introduce cohesion masks and coupling masks. Particularly, cohesion masks enhance *intra-module attention* by encapsulating lines with similar dependency patterns, thereby preserving semantic consistency within functionally related code units; while coupling masks regulate *inter-module attention* by selectively integrating dependency-aware context, guided by relevance scores derived from dependency encodings. This dual-mask mechanism balances local coherence with global coupling, enabling the transformer to maintain stable, context-aware attention at the repository scale while focusing on functionally relevant dependencies across module boundaries.

Benefits. The decoupling of local semantics from global dependencies enables COREGEN to preserve contextual information through context dependency modeling, while cohesion-coupling aware attention maintain stable distributions by restricting attention to functionally relevant code rather than dispersing across entire repositories.

COREGEN offers four key advantages over LCG and RAG approaches. First, by leveraging localized semantic representations with dependency encoding, it enables effective long-range modeling without relying on positional bias (addressing **L1**). Second, the structured attention hierarchy maintains stable attention distributions across all stages, significantly reducing perplexity degradation observed in extended context scenarios (addressing **L2**). Third, unlike the random fragmentation adopted in current RAG implementations, COREGEN operates at the granularity of natural semantic units (i.e., code lines) and preserves semantic continuity through a hierarchical design which maintains local coherence based on semantic feature modeling and ensures inter-module continuity across module boundaries (addressing **L3**). Fourth, rather than retrieving context via embedding similarity, COREGEN learns common dependency patterns through dictionary learning and identifies relevant structures via attention-based decomposition. This shifts the paradigm from similarity retrieval to dependency-aware feature matching (addressing **L4**).

Together, these advantages demonstrate that COREGEN overcomes the limitations of conventional NLP approaches by incorporating mechanisms designed for software engineering. Notably, this design retains the fundamental Transformer architecture while seamlessly embedding modular software engineering principles into the model. As a result, COREGEN can flexibly function either

as a novel attention mechanism within retrained models or as a dependency-aware contextual management framework, ensuring broad adaptability. This underscores the potential of purpose-built approaches to advance the field of AI for Software Engineering and foster a paradigm shift in its development.

Evaluation. We evaluate COREGEN on repository-level code generation and language modeling tasks using both a model trained from scratch and three representative pretrained code LLM families. Specifically, we first train COREGEN-0.2B from scratch to validate the effectiveness of the proposed attention mechanism in controlled settings. We then assess its generalizability by integrating COREGEN as an attention manager into three widely used pretrained code models, i.e., CodeLlama-7B [49], Qwen-2.5-Coder-7B [19], and DeepSeek-Coder-6.7B [15]. This setup allows us to evaluate COREGEN across different parameter scales, pretraining corpora, and code model families.

For COREGEN-0.2B trained from scratch, results on the RepoBench dataset show notable improvements over the baseline: it achieves Exact Match (EM) scores of 5.18 vs. 3.88 (cross-file first), 18.98 vs. 16.70 (cross-file random), and 18.28 vs. 18.26 (in-file) compared to the best baseline. On LongBench, COREGEN-0.2B scores 14.29 vs. 13.74 on LCC and 8.90 vs. 7.63 on RepoBench-P.

Similar improvements are consistently observed across all three pretrained code LLMs. Compared with their corresponding backbone models, COREGEN improves the average EM from 23.17 to 28.32 on CodeLlama-7B (+22.2%), from 25.70 to 30.71 on Qwen-2.5-Coder-7B (+19.5%), and from 26.02 to 30.74 on DeepSeek-Coder-6.7B (+18.1%). These improvements are consistent across all repository-level benchmarks, demonstrating that the effectiveness of COREGEN generalizes across pretrained code model families rather than being specific to any particular backbone.

On the language modeling task, COREGEN-0.2B (trained on 1k and tested up to 8k tokens) maintains stable performance with an average perplexity of 2.04, while existing methods degrade severely (e.g., RoPE increases from 2.29 to 98.15). These results demonstrate COREGEN's robustness and effectiveness across both code generation and language modeling tasks.

Structure. Section 2 gives the preliminary. Section 3 presents the COREGEN framework. Section 4 and Section 5 present the experimental setup and results. Section 6 discusses threats to validity. Section 7 reviews related work. Section 8 concludes the paper.

2 PRELIMINARY

In this section, we review attention mechanisms and current approaches to dependency modeling.

The fundamental challenge in repository-level code generation is to enable attention mechanism to effectively process extensive and heterogeneous contextual information [20]. Given a sequence of tokens $T = t_1, \ldots, t_m$ with corresponding embeddings $X = x_1, \ldots, x_m$, where m denotes the sequence length and $x_i \in \mathbb{R}^d$. The standard (unnormalized) attention score between any two tokens x_i and x_j $(1 \leq i, j \leq m)$ is computed as [55]

$$a_{i,j} = q_i^T k_j + M_{i,j}, \tag{1}$$

where $q_i = \mathbf{W_Q}(x_i + p_i)$ and $k_j = \mathbf{W_K}(x_j + p_j)$ are the query and key projections of embeddings x_i and x_j, respectively; p_i and p_j represent positional encodings at positions i and j, and $\mathbf{W_Q}, \mathbf{W_K} \in \mathbb{R}^{d \times d}$ are learnable weight matrices. $M_{i,j}$ denotes the entry of the causal attention mask matrix $M \in \mathbb{R}^{m \times m}$, which enforces constraints on which tokens can be attended to. Typically we use attention entropy to measure how widely an attention head distributes its focus across tokens in a sequence. High entropy means it is paying attention to tokens more evenly, i.e., the attention weights are spread out across multiple tokens rather than concentrated on just a few.

In repository-level code generation where sequence lengths substantially exceed those encountered during training, standard position encoding schemes struggle to generalize, leading to degraded performance on long-context generation tasks.

Beyond positional encoding, our prior work (CoDE [67]) explored dictionary learning [50, 54] to model contextual dependencies [67]. The observation is that dependencies in source code are usually sparse, namely, not every pair of code elements, but rather some specific components exhibit recurring (functional) dependency, where dictionary learning is particularly suitable.

Typically, sparse autoencoders (SAE) [8, 11, 39] are leveraged, which transform the model's hidden states into feature activations through an encoder, and then trigger specific dependency features in a learned dictionary. The learned dictionary effectively serves as a codebook of common dependency patterns, where each dictionary element represents a relationship. These extracted features are subsequently integrated back into the hidden states, enabling dependency-aware attention computation. Formally, this process can be formalized as

$$c_i = \mathrm{ReLU}(\mathbf{E}x_i + b), \quad d_i = \mathbf{D}c_i, \tag{2}$$

where $\mathbf{E} \in \mathbb{R}^{F\times d}$, $\mathbf{D} \in \mathbb{R}^{d\times F}$ and $\mathbf{b} \in \mathbb{R}^F$ denote the learnable parameters of SAE. Specifically, the encoder $\mathbf{E}$ maps the input x_i to a feature activation vector c_i, while d_i serves as a dependency-aware encoding, replacing the traditional positional encoding p_i. (It is implemented as $x_i \leftarrow x_i + d_i$.) To encourage sparsity in the activated features, regularization can be applied to c_i, which enhances both interpretability and efficiency of the learned representations.

This dependency-based encoding approach can more effectively capture the dependency relationships within code fragments, rather than relying solely on their positional relationships, which contributes to improved code generation quality and interpretability.

Limitations of Prior Work. Although dependency-based encoding showed promise for capturing functional relationships beyond positional cues, we identify three major issues when applied to repository-level scenarios.

First, to learn sparse and interpretable dependency features from large-scale training datasets, CoDE adopts ℓ_1 regularization as a sparsity constraint. This regularization term is, however, not aligned with the model's optimization objective for representation learning, which ultimately degrades training quality.

Second, the method lacks a hierarchical view on code structure. CoDE treats all dependencies as existing at a single level of granularity, while real repositories exhibit multiple layers of dependency (such as intra-method, inter-method and cross-file relationships). Without hierarchical modeling, coarse-grained features are overshadowed by highly specialized ones and thus remain inactive during inference [4].

Third, the method suffers from attention dispersion at repository scale. As discussed in the Introduction, when the entire repository is used as input, attention weights are distributed across a vast number of tokens without effectively distinguishing relevant from irrelevant dependencies. This dispersion increases perplexity and leads to performance degradation. Our previous work [67] further confirms that, as input length grows, generation quality decreases due to the same attention scattering effect.

3 OUR APPROACH

We propose COREGEN, a novel framework that leverages the structural properties of software repositories via local-global dependency analysis. This approach is motivated by the observation that well-designed code repositories typically exhibit high cohesion within modules and low coupling between them, as per fundamental software engineering principle.

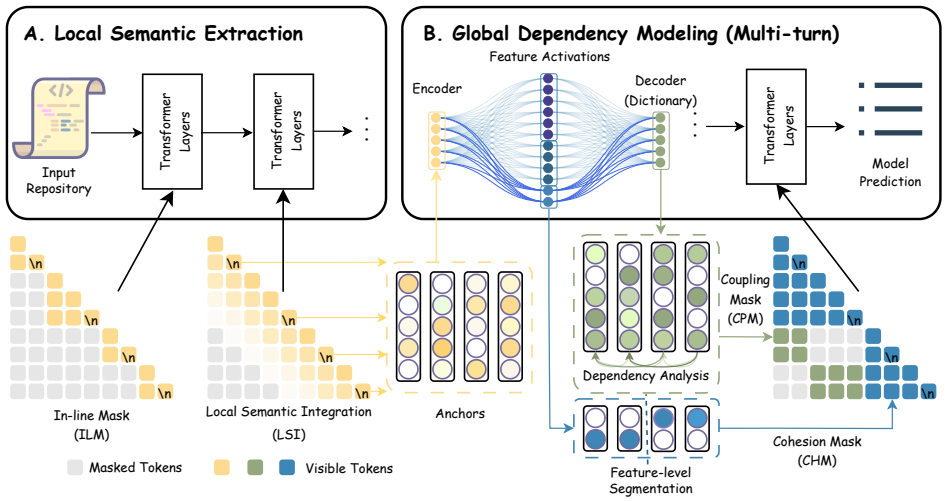


Fig. 2. An overview of the COREGEN framework.

As illustrated in Figure 2, COREGEN decomposes repository-level context modeling into two stages, i.e., *Local Semantics Extraction* and *Global Dependency Modeling*. In the first stage (cf. the left part of Figure 2 and Section 3.1), we capture fine-grained semantic information at the code-line level, based on the observation that individual lines of code often correspond to discrete data operations and thus serve as the atomic units of dependency modeling.

In the second stage (cf. the right part of Figure 2 and Section 3.2), COREGEN applies multi-level dictionary learning to capture dependencies at different granularities, based on the extracted line-level semantics from the previous step. From the dependency features, COREGEN then constructs two types of masks: cohesion masks, which enhance intra-module attention by clustering lines with similar dependency patterns, and coupling masks, which regulate inter-module attention by selectively incorporating dependency-aware context. These attention masks are used in the attention computation (as per Eqn.(1)). This dual-mask mechanism effectively balances local coherence and global interaction, ensuring stable and context-aware attention at the scale of entire code repositories.

3.1 Local Semantics Extraction

Previous research shows that code LLMs exhibit a distinctive attention pattern where disproportionately high attention weights are concentrated to specific tokens in the context, commonly referred to as *anchors* [18, 41, 57]. Extensive empirical analyses have demonstrated that newline tokens consistently function as primary anchors in code LLMs and help organize the planning of subsequent code [33, 68], serving as natural delimiters that effectively compress and encapsulate the semantic content within individual lines of source code. COREGEN leverages this observation which operates at the granularity of *code lines* rather than individual tokens.

This shift of granularity offers two important advantages. First, it aligns with the natural cognitive units (i.e., code line) that software developers use when writing and comprehending code, where individual lines typically correspond to atomic computational operations or logical statements. Second, it substantially reduces computational complexity by avoiding modeling dependencies across all tokens in extensive repository contexts.

In-line Mask. COREGEN begins with extracting semantic information contained within individual code lines. To this end, we introduce *In-line Mask* (ILM), implemented within the initial layers of our structured Transformer architecture (specifically, layers 0—1 in our experimental setup). Formally, let the sequence be segmented into lines, where line r contains tokens from position $b_{r-1}+1$ to b_r, with $1 = b_1 < \cdots < b_k = m$ representing the positions of line breaks. The mask entry $\mathrm{ILM}_{i,j}$ (for $1 \le i, j \le m$) is defined as

$$\mathrm{ILM}_{i,j} = \begin{cases} 0, & \text{if } b_{r-1} < i, j \le b_r, \\ -\infty, & \text{otherwise.} \end{cases}$$

ILM ensures that tokens can only attend to other tokens within the same line, creating a localized focus that enables the model to capture fine-grained semantics within each code line independently.

Local Semantic Integration. ILM functions independently without incorporating contextual information, which can potentially lead to semantic drift and limit the model's capacity for comprehensive dependency analysis. However, incorporating the entire context would result in increased attention entropy, meaning that the model attends to too many tokens and therefore struggles to concentrate on the most relevant context.

As a remedy, we incorporate a Local Semantic Integration mechanism into the attention score computation within the intermediate layers of from-scratch version of COREGEN (specifically, layers 2—5). This mechanism restricts attention to local context, based on the observation that local information in code is typically more informative for understanding the immediate context. By restricting attention to local context, this approach limits the growth of attention entropy over long sequences. Concretely, we adopt ALiBi [46], which introduces a static, distance-based bias to attention scores (cf. Eqn. (1)) that decays linearly with token distance.

$$a_{i,j} = q_i^T k_j - \alpha \cdot |i - j| + M_{i,j},$$

where $\alpha = 2^{-\frac{8h}{n_{\text{heads}}}}$ represents a head-specific scalar multiplier that is fixed prior to training where h denotes the index of the attention head and n_{heads} is the total number of heads. This formulation systematically decreases attention to distant tokens while preserving semantics of nearby context.

When fine-tuning pre-trained Code LLMs with COREGEN, we observe that introducing the LSI mechanism leads to strong inconsistencies with the original model's behavior, thereby degrading training quality. Moreover, since pre-trained code models have already been fine-tuned on long-text data, they inherently possess the ability to handle long-context attention entropy. As a result, we do not apply the Local Semantics Extraction mechanism to pre-trained Code LLMs.

By incorporating Local Semantic Integration (LSI), we can capture line-level semantic representations enriched with localized contextual information. This can give more robust representation of code lines while effectively controlling attention entropy in long-context scenarios.

3.2 Global Dependency Modeling

Local semantics are crucial for understanding individual code segments, but they alone are insufficient for capturing the dependencies that span distant, but functionally related, components. To bridge this gap, Global Dependency Modeling introduces two complementary mechanisms: Multi-level Dependency Encoding (MDE)and Cohesion-Coupling Aware Attention. The complete process is detailed in Algorithm 1.

3.2.1 Multi-level Dependency Encoding. We extend dictionary learning (cf. Eqn. (2)), and adopt Top-K activation [3] as regularization to select the most salient k features (Algorithm 1, line 4),

$$c_r = \mathrm{TopK}_k\left(\mathrm{ReLU}(\mathbf{E}x_{b_{r-1}} + b)\right)$$

Algorithm 1 Cohesion-coupling Aware Attention

procedure MULTILEVELENCODING
Input: $x_{b_{r-1}}$: Input representation; $\mathcal{S}$: Set of levels
Output: C: Multi-level features; $\mathcal{D}$: Multi-level dependency encodings
  $C \leftarrow \emptyset, \mathcal{D} \leftarrow \emptyset$ ▷ Initialize feature collections
  for all $s \in \mathcal{S}$ **do** ▷ Iterate across all levels
    $c_r^{(s)} \leftarrow \mathrm{TopK}_{k^{(s)}}\left(\mathrm{ReLU}(\mathbf{E}^{(s)} x_{b_{r-1}} + b^{(s)})\right)$ ▷ Extract sparse features
    $d_r^{(s)} \leftarrow \mathbf{D}^{(s)} c_r^{(s)}$ ▷ Generate multi-scale dependency encoding
    $C \leftarrow C \cup \{c_r^{(s)}\}, \mathcal{D} \leftarrow \mathcal{D} \cup \{d_r^{(s)}\}$
  end for
  return $(C, \mathcal{D})$ ▷ Return multi-level features
end procedure
procedure CO-MASK
Input: $\{(c_i^{(s)}, d_i^{(s)}),\ (c_j^{(s)}, d_j^{(s)})\}_{s\in\mathcal{S}}$: Multi-level features
Output: COM: Cohesion-coupling attention mask
  $\mathrm{COM} \leftarrow \mathbf{0}$ ▷ Initialize combined mask
  for all $s \in \mathcal{S}$ **do** ▷ Process each level separately
    $\mathrm{CHM}^{(s)} \leftarrow \mathbb{I}[\mathrm{supp}(c_i^{(s)}) = \mathrm{supp}(c_j^{(s)})] \cdot 0 + (1 - \mathbb{I}[\cdot]) \cdot (-\infty)$ ▷ Cohesion mask
    $\mathrm{CPM}^{(s)} \leftarrow \mathrm{supp}\left(\mathrm{TopK}_k\left(d_i^{(s)T} W_Q^T W_K d_j^{(s)}\right)\right)$ ▷ Coupling mask
    $\mathrm{COM} \leftarrow \min\left(\mathrm{COM}, \max(\mathrm{CHM}^{(s)}, \mathrm{CPM}^{(s)})\right)$ ▷ Aggregate masks
  end for
  return COM ▷ Return the fused multi-level mask
end procedure

where $\mathrm{TopK}_k(\cdot)$ returns a sparse vector retaining only the k largest entries while setting others to zero. The reason not to use the normal ℓ_1 regularization is that previous research [3, 22, 47] has shown that incorporating an ℓ_1 regularization term induces shrinkage, resulting in a systematic underestimation of the magnitudes of active features. Moreover, the ℓ_1 penalty is not inherently compatible with the cross-entropy loss function used for data fitting, which may adversely impact the effectiveness of the training process. In contrast, Top-K activation avoids additional loss terms during training, resulting in improved performance.

In addition, repository-level code dependencies exhibit inherent complexity across multiple granularities (e.g., intra-method, inter-method and cross-file relationships), as illustrated in Figure 2. We introduce a multi-level dependency encoding approach (Algorithm 1, lines 1–9) [4], which learns and optimizes feature representations across different levels by using varying activation frequencies. It can effectively capture the hierarchical nature of dependencies within code repositories. Let $\mathcal{S}$ be the set of feature levels corresponding to different dependency granularities. For each $s \in \mathcal{S}$,

$$c_r^{(s)} = \mathrm{TopK}_{k^{(s)}}\left(\mathrm{ReLU}\left(\mathbf{E}^{(s)} x_{b_{r-1}} + b^{(s)}\right)\right)$$
$$d_r = \sum_{s\in\mathcal{S}} d_r^{(s)} = \sum_{s\in\mathcal{S}} \mathbf{D}^{(s)} c_r^{(s)}$$

where $k^{(s)}$ denotes the number of activated features allowed for each feature level.

In dictionary learning, the dictionary size s controls the model's representational capacity. Smaller dictionaries capture coarse, recurring patterns, whereas larger dictionaries represent finer-grained dependency patterns. Varying s therefore induces a hierarchical representation: coarse levels group structurally relevant code, while finer levels distinguish precise functional

relationships. By modeling dependencies across multiple granularities, our approach captures both coarse-grained and fine-grained structures, enabling more accurate dependency modeling and improving robustness to out-of-distribution cases.

3.2.2 Cohesion-coupling Aware Attention.

Cohesion Mask. In software development, cohesion refers to the degree to which elements within a module are functionally related. In code repositories, cohesive local context plays a critical role in enabling accurate semantic understanding. This is because functionally related code fragments often exhibit similar syntactic structures, API usage patterns and data-flow behaviors.

To effectively capture cohesive structures within a repository, we define the cohesion between code fragments based on the similarity of their dependency features. Fragments that share the same dependencies are likely to serve related functions or participate in similar workflows, indicating a strong functional relationship. This perspective aligns closely with the goal of dictionary learning, which seeks to extract and represent the most salient features from input data. The distribution of these features often corresponds to cohesive functional units within the codebase.

Based on this insight, we introduce Cohesion Mask (CHM) (Algorithm 1, line 13), which guides the model to focus attention within these cohesive units. The construction of CHM is defined as follows:

$$\mathrm{CHM}_{i,j}^{(s)} = \begin{cases} 0, & \text{if } \mathrm{supp}(c_i^{(s)}) = \mathrm{supp}(c_j^{(s)}), \\ -\infty, & \text{otherwise.} \end{cases}$$

where $\mathrm{supp}(\cdot)$ denotes the support of a vector, i.e., the set of indices of its non-zero elements, indicating the positions of activated features. c_i and c_j represent the dependency encodings of the code lines to which x_i and x_j belong, respectively. This mask enables the model to attend to tokens that share the same activated dependency features, thereby preserving continuity between local and contextual information and reinforcing semantic coherence within cohesive code elements.

Coupling Mask. While cohesion addresses dependencies within modules, coupling represents critical inter-module dependencies. It quantifies the extent to which one module relies on others to function correctly. To assess inter-module dependencies, we first analyze the attention mechanism when dependency encodings are incorporated. The query-key interaction (cf. Eqn (1)) between two representations can be expanded to four distinct terms:

$$\begin{aligned} q_i^T k_j &= (W_Q(x_i + d_i))^T W_K(x_j + d_j) \\ &= x_i^T W_Q^T W_K x_j + x_i^T W_Q^T W_K d_j + d_i^T W_Q^T W_K x_j + d_i^T W_Q^T W_K d_j \end{aligned}$$

We observe that the term $d_i^T W_Q^T W_K d_j$ directly quantifies the degree of dependency between x_i and x_j, effectively capturing the functional coupling between different lines of code. Intuitively, d_i and d_j encode the respective dependency features of x_i and x_j, the matrix $W_Q^T W_K$ projects these features onto the attention space, and the inner product measures the similarity between the projected dependency features. As a result, this term captures the strength of functional coupling.

Leveraging this insight, we construct Coupling Mask (CPM) (Algorithm 1, line 14) to selectively focus attention on the most functionally relevant tokens across modules:

$$\mathrm{CPM}_{i,j}^{(s)} = \begin{cases} 0, & \text{if } j \in \mathrm{supp}\left(\mathrm{TopK}_k\left(d_i^{(s)^T} W_Q^T W_K d_{j'}^{(s)} \mid 1 \le j' \le i\right)\right) \\ -\infty, & \text{otherwise.} \end{cases}$$

Here, $\mathrm{supp}\left(\mathrm{TopK}_k(\cdot)\right)$ identifies the top k positions with the highest values of $d_i^{(s)^T} W_Q^T W_K d_{j'}^{(s)}$ from all targets $1 \le j' \le i$, $d_i^{(s)}$ is the extracted encoding of the level s. Intuitively, CPM enables

each position to attend to a sparse set of functionally relevant tokens across modules, thereby facilitating more effective modeling of essential cross-module dependencies. Unlike explicit hierarchical approaches (e.g., tree- or graph-based methods) that rely on predefined structural assumptions from static analysis, our multi-level dictionary design is learned end-to-end from data. This makes it structure-agnostic and language-independent, without requiring external parsers or handcrafted rules. As a result, it remains robust to incomplete or erroneous parse results caused by a fixed context window, which commonly arise when training on real-world repositories, thereby enhancing robustness and generalizability.

Since our dependency analysis spans multiple levels, we analyze code cohesion and coupling at varying levels of granularity. To this end, we iterate through each level in the multi-level dependency encoding (Algorithm 1, lines 10–15), computing level-specific cohesion and coupling masks. These masks are then aggregated to construct the combined cohesion-coupling attention mask matrix, denoted as COM (Algorithm 1, line 15):

$$\mathrm{COM} = \min_{s \in \mathcal{S}} \left(\max \left(\mathrm{CHM}^{(s)}, \mathrm{CPM}^{(s)} \right) \right)$$

where $\mathrm{CHM}^{(s)}$ and $\mathrm{CPM}^{(s)}$ represent the cohesion and coupling masks computed for the level s, respectively. We use max aggregation to enable each token pair to incorporate the commonly attended code fragments across levels, ensuring that critical information at different granularity is effectively integrated into the attention mechanism. The inner max takes the union of CHM and CPM at each level, allowing a token to be attended to if it is either cohesive or coupled. The outer min enforces that a token must be permitted across all levels to remain in the final attention, effectively filtering out tokens that are only incidentally relevant at a single granularity. By fusing these multi-level masks, COREGEN effectively balances the preservation of intra-module coherence with the selective modeling of inter-module dependencies, ultimately enhancing dependency-aware code generation.

4 EXPERIMENTAL SETUP

To assess the effectiveness of COREGEN, we conduct experiments designed to address the following research questions (RQs).

- **RQ1:** How effective is COREGEN in repository-level code generation tasks compared to baseline approaches?
- **RQ2:** How does COREGEN perform compared to state-of-the-art LCG methods in terms of perplexity on long code sequences?
- **RQ3:** What are the contributions of the components within COREGEN to its performance on repository-level generation?

Implementation. To evaluate the effectiveness of COREGEN, we conduct experiments under both pre-training and fine-tuning settings. We first train COREGEN-0.2B from scratch to evaluate the proposed attention mechanism in a controlled pre-training setting. As pre-training multiple large code LLMs from scratch would require prohibitively large computational resources, we further assess the generalizability of COREGEN by integrating it into three representative pretrained code LLMs, i.e., CodeLlama-7B [49], Qwen-2.5-Coder-7B and DeepSeek-Coder-6.7B. Each model is then fine-tuned using the same two-stage training pipeline described below. Although fine-tuning cannot fully substitute end-to-end pre-training at this scale, it provides a practical and widely adopted protocol for evaluating whether architectural modifications generalize across diverse pretrained code model families.

We train COREGEN-0.2B from scratch on 20 billion tokens of Python code from the StarCoder dataset [28]. The model consists of 12 transformer layers, with a hidden size of 768 and 8 attention heads. Training is performed with a global batch size of 128 (16 × 8) and a learning rate of 5e-4. Cohesion–coupling masks are applied from layer 6 to layer 12, and the number of coupling tokens is set to 512. This layer assignment is designed to reflect how developers comprehend code. ILM (layers 0–1) treats each code line as the minimal logical unit and extracts its semantics independently, ensuring the accuracy of subsequent dependency modeling. LSI (layers 2–5) integrates local contextual information into these representations. Cohesion-Coupling Masks (layers 6–11) then perform global dependency matching on the enriched representations. Although ILM is restricted to the initial layers, the Transformer's residual connections propagate the independent line-level representations into deeper layers, preserving their contribution throughout the network.

Given the limited computational resources, we adopt the same two-stage fine-tuning pipeline for all pretrained code LLMs, including CodeLlama-7B, Qwen-2.5-Coder-7B, and DeepSeek-Coder-6.7B. In the first stage, the objective is to enable the model to learn code dependency information. The dictionary in COREGEN is trainable, while all original model weights are frozen, and training is performed on 250M tokens. The cohesion–coupling mask is not enabled in this stage, since altering the model's attention pattern at this point would disrupt the behavior learned during pre-training. In the second stage, the goal is to adapt the model to the modified attention pattern. We activate the cohesion–coupling mask and apply LoRA fine-tuning [17] with rank 32 to the attention projection matrices W_Q, W_K, W_V and W_O, using 25M tokens. In both stages, we use a batch size of 8 (8×1) and a learning rate of $5e-5$. Cohesion–coupling masks are applied from layer 16 to layer 32 for CodeLlama-7B and DeepSeek-Coder-6.7B, and from layer 14 to layer 28 for Qwen-2.5-Coder-7B, while the number of coupling tokens is set to 900 for all pretrained models. At all scales, COREGEN uses dictionary sizes of 128, 1,024, 16,384, with 1, 2, 8 activated features per level. All models are trained with a sequence length of 1,024.

Datasets and Tasks. We evaluate COREGEN on three datasets to examine its performance across various repository-level code generation tasks.

(1) **RepoBench** [36]: this comprises three code generation tasks designed to evaluate cross-file and in-file reasoning capabilities.
   - `cross_file_first`: The target line is the first in-file usage of a snippet that originates from a different file.
   - `cross_file_random`: The target line uses a cross-file snippet but is not its first in-file appearance.
   - `in_file`: The target line is based solely on intra-file content.

(2) **LongBench** [1]: We consider two tasks from the LongBench dataset to validate performance on extended code contexts.
   - `LCC`: this task is to predict the next line given a single-file code context.
   - `RepoBench-P`: this task is to predict the next line of code based on context spanning multiple files with cross-file dependencies.

(3) **StarCoder-Long** [28]: We construct this benchmark from the StarCoder corpus to evaluate perplexity under long-context settings. It comprises 2,000 sequences, each with 8,192 tokens. To avoid any train-test overlap, all sequences are sampled from repositories held out from the training split, i.e., no repository contributing to StarCoder-Long appears in the pre-training data.

Baselines. We compare COREGEN against a diverse set of baselines, categorized into two groups.

(1) **Long Context Generation (LCG) Methods:**

- **RoPE** [53], which uses rotary position embedding to model relative positional distances.
- **ALiBi** [46], which introduces position-aware attention by incorporating a distance-based bias in the attention score computation.
- **CoDE** [67], our prior work which applies dictionary-based semantic matching to encode dependencies.

(2) **Retrieval-Augmented Generation (RAG) Methods:**

- **RAG** [25], which augments generation by a single-pass retriever that selects similar context snippets from repository.
- **RepoCoder** [64], which employs an iterative retrieve-then-generate pipeline.
- **RepoFuse** [31], which fuses retrieved analogy context and structural rationale context.
- **RLCoder** [58], which employs reinforcement learning to train a retriever for retrieval-augmented code generation.

To ensure fair comparison, all LCG baselines were trained with the same model architecture and training corpus. For RAG methods, we adopt RoPE-based models as the foundation, given RoPE's widespread adoption in the community. However, since RoPE suffers from limited length extrapolation capabilities, we restrict the context window of these RoPE-based models to their training length of 1,024 tokens during repository-level code generation evaluation. Due to the incompatibility between different positional encoding schemes, in evaluating 7B models, we only compare them against their respective base models and RAG-based methods. Potential benchmark contamination is discussed in the Section 6.

Evaluation Metrics. We adopt the following metrics to assess the performance of COREGEN and baseline methods.

- **Exact Match (EM)**, which is the percentage of predictions that exactly match the ground-truth code line token-by-token.
- **Edit Similarity (ES)**, which evaluates the token-level similarity between the generated and reference code. Higher scores denote better alignment with the ground-truth.
- **CodeBLEU** (**CB**, [48]), which is an extension of BLEU tailored for code, incorporating both syntactic and semantic structures. It is used to assess the quality of generated outputs.
- **Dependency Invocation** (**DI**) measures the proportion of cross-file dependency invocations in the generated code that correctly match the ground-truth dependencies, evaluating the model's ability to produce functionally correct cross-module references.
- **Perplexity** serves as a direct and widely used metric for evaluating model performance. On StarCoder-Long, it measures the model's ability to perform autoregressive prediction. A lower perplexity score reflects stronger language modeling capabilities.

5 EXPERIMENTAL RESULTS

5.1 Results for RQ1-3

RQ1: Effectiveness on Repository-Level Code Generation. We evaluate COREGEN on concrete repository-level code generation tasks, against both LCG and RAG methods. Table 1 presents the performance comparison on the RepoBench and LongBench datasets using EM, ES and CB as evaluation metrics.

For the cross-file task, COREGEN-0.2B achieves the highest scores in both `cross_file_first` (EM: 5.18, ES: 42.23, CB: 13.69) and `cross_file_random` (EM: 18.98, ES: 56.75, CB: 26.45) scenarios. Compared to RAG-based methods, the improvement is from the use of dependency-aware segmentation (as opposed to random fragmentation in RAG). Furthermore, the multi-level dependency encoding captures functional relationships that span across files, enabling more accurate cross-file code generation.

Table 1. Performance comparison on repository-level code generation benchmarks. The best results are highlighted in **boldface**.

Method	RepoBench									LongBench						Average			
	cross_file_first			*cross_file_random*			*in_file*			*LCC*			*RepoBench-P*			*Overall*			
	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB	DI
Long Context Generation Methods																			
RoPE	2.28	40.10	12.52	16.24	54.36	24.15	17.78	55.04	24.99	13.74	53.14	26.33	6.36	48.25	17.79	11.28	50.18	21.16	16.27
ALiBi	1.52	35.64	9.86	5.50	41.55	13.65	6.22	44.32	14.21	3.30	39.95	13.22	2.54	39.86	11.43	3.82	40.26	12.47	6.06
CoDE	3.88	41.48	13.32	16.70	55.27	24.89	17.98	55.32	24.88	13.19	52.81	26.17	7.63	48.81	17.92	11.88	50.74	21.44	19.30
Retrieval Augmented Generation Methods																			
+ RAG	3.78	41.42	13.11	15.98	54.14	23.88	18.26	**55.37**	24.85	13.19	52.80	25.81	7.20	48.02	17.71	11.68	50.35	21.07	17.06
+ RepoCoder	3.82	41.51	13.18	15.96	54.23	23.92	18.18	55.21	24.92	13.74	53.19	26.08	7.63	**50.71**	17.92	11.87	50.97	21.20	17.17
+ RepoFuse	3.74	41.64	13.27	15.86	54.17	23.81	17.92	55.25	24.75	12.64	53.37	26.73	6.78	49.28	17.88	11.39	50.74	21.29	18.07
+ RLCoder	4.12	41.31	13.21	16.48	54.52	24.12	17.82	54.29	24.71	13.74	53.82	27.17	8.05	50.32	17.77	12.04	50.85	21.40	18.74
COREGEN-0.2B	**5.18**	**42.23**	**13.69**	**18.98**	**56.75**	**26.45**	**18.28**	54.72	**25.49**	**14.29**	**54.03**	**28.15**	**8.90**	50.56	**18.87**	**13.13**	**51.66**	**22.53**	**21.55**
Fine-Tuned Code Models																			
CodeLlama-7B	16.54	52.27	24.32	31.18	65.52	36.30	29.86	62.19	33.28	20.88	58.91	37.29	17.37	56.57	25.12	23.17	59.09	31.26	33.45
+ RAG	17.22	53.47	25.01	32.82	67.37	36.98	28.90	61.71	32.67	21.98	59.32	38.21	18.22	57.31	26.48	23.83	59.84	31.87	33.78
+ RepoCoder	18.68	55.28	26.87	33.62	68.77	37.59	29.28	60.89	32.52	22.53	60.74	38.98	19.06	57.94	26.85	24.63	60.72	32.56	34.23
+ RepoFuse	18.28	54.47	26.03	33.32	67.12	36.61	28.78	59.55	31.37	21.98	59.18	38.37	18.64	57.39	26.54	24.20	59.54	31.78	33.89
+ RLCoder	18.78	56.23	27.12	33.96	68.69	37.04	29.32	61.29	32.27	23.08	62.18	39.35	19.49	58.33	27.21	24.93	61.34	32.60	34.90
+ COREGEN	**22.08**	**61.83**	**30.53**	**38.46**	**73.47**	**43.41**	**32.08**	**65.66**	**36.19**	**25.27**	**67.41**	**42.56**	**23.73**	**61.37**	**29.29**	**28.32**	**65.95**	**36.40**	**42.20**
Qwen-2.5-Coder-7B	16.02	52.16	23.28	30.04	65.28	36.52	29.98	60.89	33.88	27.47	68.18	43.2	25.00	62.38	30.15	25.70	61.78	33.41	35.79
+ RAG	17.22	52.39	24.92	31.92	66.20	36.81	30.68	61.27	34.89	29.12	68.07	43.49	25.85	62.97	30.88	26.96	62.18	34.20	37.01
+ RepoCoder	17.42	53.99	25.03	32.18	66.59	36.02	31.44	62.21	35.23	30.77	68.95	43.27	25.42	62.28	30.69	27.45	62.80	34.05	36.97
+ RepoFuse	17.88	54.27	24.89	32.56	66.82	36.58	31.38	63.02	35.01	30.77	68.67	42.90	24.15	62.02	29.97	27.35	62.96	33.87	36.24
+ RLCoder	18.62	54.78	26.69	33.36	65.12	36.13	32.06	63.33	35.27	32.42	68.97	43.08	26.27	63.39	31.58	28.55	63.12	34.55	37.64
+ COREGEN	**21.60**	**60.11**	**30.01**	**36.22**	**68.02**	**43.82**	**33.16**	**64.74**	**39.05**	**34.62**	**70.37**	**46.15**	**27.97**	**65.16**	**33.54**	**30.71**	**65.68**	**38.51**	**43.83**
Deepseek-Coder-6.7B	15.98	53.89	23.36	30.36	66.39	35.57	28.80	60.21	34.02	29.12	69.21	44.87	25.85	63.84	31.91	26.02	62.71	33.95	36.42
+ RAG	16.32	52.14	23.68	31.18	65.68	36.47	29.52	61.39	35.17	29.12	70.89	45.24	25.85	64.21	31.97	26.40	62.86	34.51	36.63
+ RepoCoder	16.68	52.65	23.92	32.08	66.43	36.25	30.24	61.87	35.20	31.32	71.71	45.92	26.69	64.73	32.04	27.40	63.48	34.67	37.06
+ RepoFuse	16.42	52.07	23.17	31.70	65.42	36.11	30.06	62.15	35.01	31.32	71.32	45.66	26.27	64.53	31.59	27.15	63.10	34.31	36.77
+ RLCoder	17.98	52.79	24.04	32.32	67.02	37.29	31.88	**65.79**	35.77	32.97	72.55	46.03	27.54	65.58	32.65	28.54	64.74	35.16	37.92
+ COREGEN	**20.46**	**61.10**	**26.93**	**34.26**	**69.59**	**38.81**	**32.52**	65.55	**36.56**	**36.81**	**74.22**	**50.55**	**29.66**	**67.50**	**33.63**	**30.74**	**67.59**	**37.30**	**44.37**

To further evaluate the generalizability of COREGEN across different code LLMs, we fine-tune three representative models, i.e., CodeLlama-7B [49], Qwen-2.5-Coder-7B [19] and DeepSeek-Coder-6.7B [15], using the same two-stage training strategy. As shown in Table 1, COREGEN improves all three model families across every benchmark. Compared with their respective backbone models, COREGEN increases the average EM from 23.17 to 28.32 (+22.2%) for CodeLlama-7B, from 25.70 to 30.71 (+19.5%) for Qwen-2.5-Coder-7B, and from 26.02 to 30.74 (+18.1%) for DeepSeek-Coder-6.7B. Similar improvements are also observed on ES and CodeBLEU across all repository-level tasks, demonstrating that the effectiveness of our cohesion-coupling attention mechanism generalizes across diverse pretrained code model families rather than being tied to a particular backbone.

Compared with retrieval-augmented baselines, COREGEN achieves the best performance for every model family. For example, on the challenging `cross_file_first` benchmark, COREGEN improves EM from 18.78 to 22.08 on CodeLlama-7B, from 18.62 to 21.60 on Qwen-2.5-Coder-7B, and from 17.98 to 20.46 on DeepSeek-Coder-6.7B, outperforming the strongest retrieval-based baseline (RLCoder). These results indicate that explicitly modeling software modularity through cohesion-coupling attention is more effective than similarity-based retrieval, regardless of the underlying pretrained code model.

Beyond text-based metrics, we further evaluate the Dependency Invocation rate (DI), which measures whether generated code correctly invokes cross-file dependencies. COREGEN consistently achieves the highest DI across all evaluated model scales and architectures. On the 0.2B model, DI increases from 18.74 for the strongest RAG baseline to 21.55. On the larger pretrained models, COREGEN achieves DI scores of 42.20 on CodeLlama-7B, 43.83 on Qwen-2.5-Coder-7B, and 44.37 on DeepSeek-Coder-6.7B, substantially outperforming the corresponding RLCoder-enhanced baselines

(34.90, 37.64, and 37.92, respectively). These results demonstrate that our dependency-driven attention mechanism improves not only lexical generation quality but also the functional correctness of cross-file dependency invocation.

The consistent improvements across all metrics highlight COREGEN's ability. Unlike RAG approaches, the cohesion-coupling masks explicitly model functional dependencies, leading to more contextually grounded predictions.

To further demonstrate the superiority of dependency modeling over similarity-based retrieval, we use the outputs of a RAG-based retriever as input to COREGEN. Table 2 shows that it results in a performance decline compared to providing COREGEN with the entire repository. This suggests that COREGEN is capable of identifying more relevant contextual fragments than RAG-based methods.

Summary for RQ1

COREGEN consistently outperforms both LCG and RAG baselines across all tasks and scales. The performance gains are particularly pronounced in cross-file scenarios, where dependency modeling is most critical.

Table 2. Performance of COREGEN combined with the RAG-based approach for repository-level code generation. The best results are highlighted in bold.

Method	RepoBench									LongBench						Average		
	cross_file_first			*cross_file_random*			*in_file*			*LCC*			*RepoBench-P*			*Overall*		
	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB	EM	ES	CB
COREGEN-0.2B	**5.18**	**42.23**	**13.69**	**18.98**	**56.75**	**26.45**	**18.28**	**54.72**	**25.49**	**14.29**	**54.03**	**28.15**	**8.90**	**50.56**	**18.87**	**13.13**	**51.66**	**22.53**
+ RAG	4.36	41.27	13.36	17.84	55.07	25.67	17.82	53.19	25.11	13.74	52.97	26.73	7.63	49.37	18.02	12.28	50.37	21.78
+ RepoCoder	4.52	41.21	13.10	18.02	55.84	26.14	17.94	53.38	25.39	**14.29**	53.27	27.32	8.05	50.24	18.34	12.56	50.79	22.06
+ RepoFuse	4.18	41.23	13.01	17.88	55.21	25.67	17.64	53.03	24.95	13.19	52.67	26.21	7.20	49.81	18.09	12.02	50.39	21.59
+ RLCoder	4.68	41.67	12.88	18.22	55.71	26.09	18.04	54.51	25.27	13.74	53.64	27.47	8.05	50.07	18.29	12.55	51.12	22.00

RQ2: Comparison with LCG Methods. In this experiment, we first evaluate the ability of COREGEN-0.2B to model code sequences of varying lengths in comparison to other LCG methods, using perplexity as the evaluation metric. This task directly reflects the model's effectiveness in long-context generation. Table 3 shows the perplexity scores of LCG approaches on the StarCoder-Long dataset across four different sequence lengths.

Table 3. Perplexity comparison at different sequence lengths on StarCoder-Long dataset (lower is better).

Method	Sequence Length				
	1024	**2048**	**4096**	**8192**	**Avg.**
RoPE	2.29	22.70	80.03	98.15	50.79
ALiBi	**2.24**	2.06	2.91	3.21	2.61
CoDE	2.26	2.05	2.57	3.72	2.65
COREGEN-0.2B	2.32	**1.98**	**1.88**	**1.99**	**2.04**

The results reveal critical insights regarding the long-context generation capabilities of different methods. RoPE exhibits severe performance degradation as sequence length increases, with perplexity rising dramatically from 2.29 at 1024 tokens to 98.15 at 8192 tokens, demonstrating poor performance in long-context generation. In contrast, ALiBi and CoDE demonstrate more stable performance across varying sequence lengths, maintaining relatively low perplexity. However,

both methods still show performance degradation as context length increases, with ALiBi rising from 2.24 to 3.21 and CoDE from 2.26 to 3.72. This is attributed to the suboptimal local attention and attention entropy management introduced by their respective design principles in long-context scenarios.

COREGEN-0.2B achieves the best overall performance with an average perplexity of 2.04, demonstrating remarkable consistency across all sequence lengths. Notably, COREGEN-0.2B maintains stable performance even at the longest sequence length (8192 tokens), with a perplexity of only 1.99. This stability is due to its ability to maintain balanced attention entropy distributions through our hierarchical local-global modeling approach.

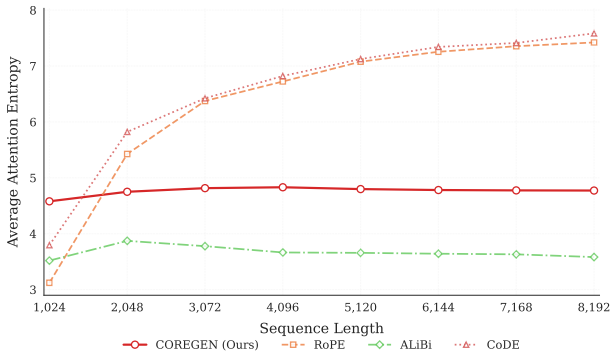


Fig. 3. Attention entropy across sequence lengths.

Figure 3 shows the attention entropy, which reflects the stability of the model's attention across various sequence lengths. RoPE and CoDE exhibit increasing entropy as the sequence length grows, indicating more diffused attention that hinder the model's ability to focus on critical context. In contrast, ALiBi maintains a stable entropy profile by design, enforcing local attention bias through its linear position penalty. However, this strict locality results in an inability to capture long-range dependencies effectively, leading to a gradual increase in perplexity. COREGEN-0.2B, on the other hand, maintains a balanced entropy across all sequence lengths due to its cohesion-coupling aware attention, allowing it to allocate attention flexibly and maintain stable, low perplexity even with context length of 8,192.

Summary for RQ2

COREGEN maintains stable perplexity (average 2.04) as the context length increases from 1,024 to 8,192 tokens, whereas all baselines exhibit noticeable degradation. Attention entropy analysis suggests that this stability stems from more balanced attention distributions, preserved by the cohesion-coupling aware attention mechanism.

RQ3: Ablation Study.

To investigate the contribution of each component in COREGEN to the overall performance, we conduct an ablation study by systematically removing individual components. Table 4a presents the results on the RepoBench dataset, where ILM, LSI, CHM and CPM denote the *In-line Mask*, *Local Semantic Integration*, *Cohesion Mask* and *Coupling Mask*, respectively.

The removal of ILM results in the most significant performance degradation across all tasks, as ILM is crucial for capturing the semantics of individual code lines, which enables effective

Table 4. Ablation study results on COREGEN-0.2B.

(a) Results on components.

Variants	RepoBench								
	cross_file_first			*cross_file_random*			*in_file*		
	EM	ES	CB	EM	ES	CB	EM	ES	CB
COREGEN	**5.18**	**42.23**	**13.69**	**18.98**	**56.75**	**26.45**	**18.28**	54.72	**25.49**
w/o ILM	2.12	38.27	10.88	10.90	48.83	17.25	12.42	50.23	19.77
w/o LSI	4.38	42.08	13.35	17.22	55.98	26.01	17.66	54.02	24.87
w/o CHM	**5.18**	42.09	13.41	18.72	56.31	25.98	18.08	52.17	24.86
w/o CPM	3.98	41.23	12.79	16.22	54.37	24.63	17.04	**54.87**	24.31
w/o MDE	4.32	41.72	12.81	16.72	54.67	24.82	17.86	53.89	24.91

(b) Results on dictionary sizes.

Dictionary Sizes	RepoBench								
	cross_file_first			*cross_file_random*			*in_file*		
	EM	ES	CB	EM	ES	CB	EM	ES	CB
{16, 128, 2,048}	5.10	41.27	13.01	17.94	55.68	26.02	17.20	53.12	24.70
{32, 256, 4,096}	5.02	41.81	13.27	18.12	56.12	26.19	17.40	53.28	24.90
{128, 1,024, 16,384}	5.18	42.23	13.69	18.98	56.75	26.45	18.28	54.72	25.49
{256, 2,048, 32,768}	5.24	42.49	14.01	**19.12**	56.92	26.76	18.20	54.78	25.61
{512, 4,096, 65,536}	**5.66**	**43.30**	**14.70**	19.08	**57.23**	**26.81**	**19.12**	**55.17**	**26.03**

(c) Results on activated features.

Features	RepoBench								
	cross_file_first			*cross_file_random*			*in_file*		
	EM	ES	CB	EM	ES	CB	EM	ES	CB
{1, 1, 1}	4.98	41.87	13.31	17.98	55.85	26.31	17.62	53.87	24.84
{1, 2, 8}	5.18	**42.23**	**13.69**	**18.98**	**56.75**	**26.45**	**18.28**	54.72	**25.49**
{4, 8, 32}	**5.22**	42.08	13.43	18.50	55.88	25.92	18.12	**54.78**	25.31
{32, 64, 256}	4.82	41.78	13.20	18.22	55.33	25.63	17.90	54.02	24.98

(d) Results on coupling tokens.

Tokens	RepoBench								
	cross_file_first			*cross_file_random*			*in_file*		
	EM	ES	CB	EM	ES	CB	EM	ES	CB
{128}	4.86	41.01	13.25	17.90	54.37	26.02	17.62	54.05	25.31
{256}	5.04	**42.33**	13.27	18.52	56.12	26.07	**18.30**	53.86	**25.71**
{512}	5.18	42.23	**13.69**	**18.98**	**56.75**	**26.45**	18.28	**54.72**	25.49
{768}	**5.26**	42.31	13.56	18.82	56.12	25.89	18.04	53.97	25.44

dependency modeling. Removing LSI leads to moderate performance degradation, especially in cross-file tasks, indicating that integrating local context is vital for enhancing semantic representations. Additionally, its removal alters attention entropy, further reducing model effectiveness. Eliminating CHM causes minimal impact on exact match scores but leads to noticeable drops in Edit Similarity and CodeBLEU, suggesting that cohesion improves generation fluency and structural quality rather than exact token-level matching. The removal of CPM results in consistent performance degradation across all tasks, with particularly pronounced effects in cross-file scenarios. This highlights the benefits of focusing on key contextual fragments for long-context code generation. Removing MDE causes notable degradation in cross-file tasks. A single-level dictionary cannot simultaneously capture coarse-grained structural patterns and fine-grained dependency relationships. In particular, coarse features tend to lose coverage on specific subcases that would otherwise be captured by specialized representation.

To evaluate the scaling capability of dictionary size, we conduct experiments by progressively enlarging the dictionary size, as shown in Table 4b. The empirical results demonstrate that model performance improves consistently with the expansion of the dictionary, thereby providing strong evidence that COREGEN's scalability. Although larger dictionaries (e.g., {512, 4,096, 65,536}) yield marginal improvements, we adopt the configuration {128, 1,024, 16,384} to strike an optimal balance between computational efficiency and generation quality.

We further investigate the sensitivity of two key TopK parameters, i.e., (1) the number of activated features per level, and (2) the number of coupling tokens. Table 4c presents the effect of varying the number of activated features. The configuration {1,1,1} performs poorly because overly sparse activations limit the dictionary's capacity to capture diverse dependency patterns. The configuration {32,64,256} also degrades performance, as excessive activations produce highly correlated features and increase the risk of overfitting. Performance remains stable within the range from {1,2,8} to {4,8,32}, indicating that the method is robust to this parameter.

Table 4d reports the effect of varying the number of coupling tokens. When the number is small (e.g., 128), cross-module information flow is limited. When it is too large (e.g., 768), the model's ability to learn discriminative context selection patterns is weakened. Notably, the performance

Table 5. Multi-level dependency encoding: Feature 104 (typed assignments) and its contextual refinements.

Feature	Explanation of individual feature	Explanation of combined features
D0_F104	**Generic typed value binding:** Assigning primitive values (int, str, bool) to variables	**Base Feature: –**
+ L1_F125	**Value propagation:** Return statement processing and output handling	**Base + Value propagation** = Ensuring type consistency across function boundaries
+ L1_F308	**Configuration patterns:** System-wide constant definitions	**Base + Configuration patterns** = Establishing immutable typed constants for application settings
+ L1_F451	**Object construction:** Class instantiation patterns	**Base + Object construction** = Setting typed attributes during instance initialization
+ L1_F647	**Function signatures:** Parameter declarations and contracts	**Base + Function signatures** = Defining type-constrained function parameters with defaults
+ L1_F651	**Conditional selection:** Runtime-dependent assignments	**Base + Conditional selection** = Type-preserving conditional value selection

gap between 256 and 512 is small, which is consistent with our prior finding [68] that code models do not require attention over a large number of tokens to achieve accurate predictions. Among these configurations, 512 provides the best overall trade-off.

Summary for RQ3

All components contribute positively to overall performance, with ILM and CPM having the most significant impact, as they jointly form the foundation for line-level semantic extraction and cross-module dependency identification. Sensitivity analyses further demonstrate that COREGEN is robust to hyperparameter variations within reasonable ranges.

5.2 Multi-level Feature Interpretability

To investigate the semantic compositionality learned by COREGEN-0.2B, we conduct an in-depth analysis of the hierarchical dependency features. Technically, we adopt the auto-interpretation protocol [2] for mechanistic interpretability. We first collect samples from the StarCoder dataset and feed them into the model, logging the activated features together with their corresponding code lines. For each dependency feature, we then sample multiple code snippets (with context) and feed them to GPT-4o. Based on these examples, GPT-4o infers the activation patterns of these features and produces human-interpretable explanations for each feature's semantic role and compositional behavior. Some results are presented in Tables 5 and 6, which exemplify how COREGEN systematically constructs complex programming patterns through the interaction between coarse-grained syntactic structures (Level 0) and fine-grained operational contexts (Level 1). Specifically, we present explanations for features at different levels (e.g., L0_F104 refers to the 104th Level 0 feature within a dictionary) and their compositional relationships.

Specifically, the typed assignment specialization (L0_F104) shows how generic typed value bindings extend to practical patterns like value propagation, configuration, object construction, function signatures, and conditional selection. Similarly, the container initialization specialization

Table 6. Multi-level dependency encoding: Feature 105 (structured data initialization) and its contextual refinements.

Feature	Explanation of individual feature	Explanation of combined features
L0_F105	**Abstract container construction:** Creating lists, tuples, dictionaries with formatted content	**Base Feature: –**
+ L1_F83	**Caching mechanisms:** Memoization and result storage	**Base + Caching mechanisms** = Constructing data structures to manage resource
+ L1_F651	**Traversal logic:** Sequential element processing	**Base + Traversal logic** = Creating new storage architectures optimized for loop-based data structures
+ L1_F761	**Mutation operations:** In-place updates and state changes	**Base + Mutation operations** = Initializing containers explicitly designed for runtime modifications
+ L1_F954	**Tabular data operations:** Column/row indexing and slicing patterns	**Base + Tabular data operations** = Constructing 2D data structures for matrix operations
+ L1_F1003	**Nested assignments:** Multi-level variable bindings	**Base + Nested assignments** = Building tree-structured data with parent-child relationships

(L0_F105) illustrates how abstract container creation underpins efficient caching, traversal, mutation, tabular operations, and hierarchical data organization.

Key Insights. The typed assignment feature L0_F104 shows how simple mechanisms like variable binding gain expressiveness when combined with context-specific intentions such as propagation, configuration, or conditional logic. Similarly, the structured data initialization feature L0_F105 illustrates how general container creation supports tasks like caching, traversal, and hierarchical nesting. This layered interaction enables COREGEN to connect low-level syntax with high-level goals, producing representations that are both flexible and grounded in real coding practices.

5.3 Case Study

In Figure 4, we present a representative example from the RepoBench dataset to demonstrate how COREGEN-0.2B operates in a real-world code repository. This figure illustrates how COREGEN's first attention head processes Level 1 features while preparing to predict the next token. As described in Section 3.2.2, COREGEN uses the CHM and CPM modules to apply attention masks that restrict what parts of the code the model can attend to. The visualization clearly reveals which tokens remain visible after masking, highlighting how COREGEN preserves relevant context and concentrates its attention on the code segments most critical for accurate generation.

This example reflects a common yet complex scenario in large-scale repositories: integrating and invoking functions scattered across multiple files. Here, the model is expected to generate a function call to `make_scheduler`. To do so, the model must simultaneously capture two types of context: the local context, which defines key parameters and operations (e.g., the optimizer constructed by `make_optimizer` given in the `Trainer` class in `trainer.py`), as well as the cross-file context, which specifies the expected input arguments and output of `make_scheduler` as defined separately in `lr_scheduler.py`.

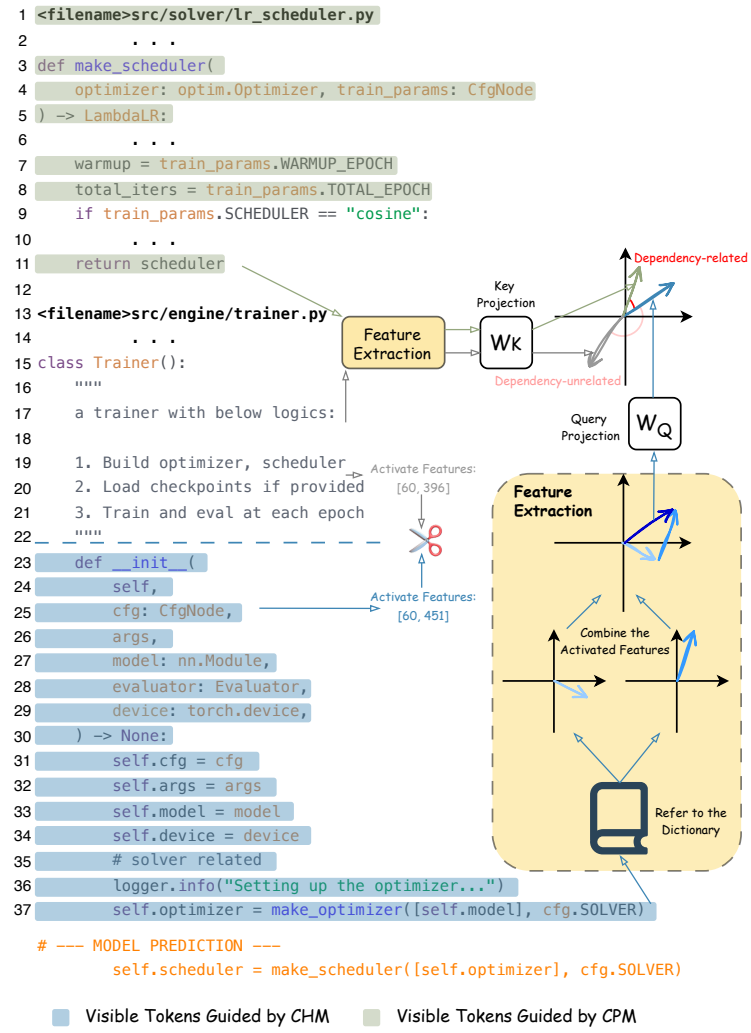


Fig. 4. A case study of COREGEN-0.2B. We demonstrate how COREGEN-0.2B processes code repository contexts and present a two-dimensional visualization for clearer illustration.

The visualization results demonstrate how COREGEN processes code in a way that aligns with a human developer. The CHM module clusters and maintains local dependencies based on extracted dependency features, ensuring that immediate variables, parameter scopes, and nearby logic are correctly identified. Meanwhile, the CPM projects the extracted dependency groups into the attention representation space and analyzes their interrelations to accurately identify essential cross-file connections. Together, these components enable the model to generate code that faithfully preserves the intended data flow, producing precise method invocations, correct input bindings, and valid return values.

In contrast, methods relying solely on LCG tend to overlook long-range dependencies and often fail to resolve the correct signature (e.g., input and output) of the target method. Similarly, RAG-based approaches retrieve functions such as `make_optimizer` purely based on surface similarity, which only provides limited support for precise and contextually correct code generation.

6 THREATS TO VALIDITY

In this section, we identify and discuss several potential threats to the validity of our study and the steps taken to mitigate them.

Internal Validity. The primary threat is from the implementation details of the compared approaches. To mitigate this, we carefully followed the original specifications of the baseline approaches and verified our implementations against the reported results.

External Validity. One potential threat is the representativeness of our evaluation datasets. While StarCoder, RepoBench and LongBench are widely recognized benchmarks, they may not cover all possible real-world repository-level code generation scenarios. To address this concern, we selected diverse tasks that encompass both in-file and cross-file dependencies. Regarding potential training–test overlap, StarCoder-Long is constructed from repositories that are disjoint from the training split by design, following the standard practice of creating held-out evaluation repositories for language modeling benchmarks. For RepoBench and LongBench, we performed an exact-match overlap analysis between the evaluation benchmarks and the training data. Specifically, we extracted the ground-truth target code for each evaluation sample and performed exact line-level matching against both the pre-training corpus of COREGEN-0.2B and the fine-tuning corpus used for all 7B models. The overlap rate was computed as the percentage of evaluation target lines with an exact match in the training data. This analysis identified an overlap of only 0.06% of target lines, indicating that benchmark contamination is negligible. Additionally, our evaluation includes a 0.2B model trained from scratch and three 7B models using a two-stage strategy. The consistent improvements across both scales suggest promising scalability of our approach. However, we acknowledge that fine-tuning may not fully reveal the method's potential compared to end-to-end pre-training, which requires computational resources beyond the capacity of most academic research groups. We leave large-scale pre-training as future work.

Construct Validity. The metrics used (perplexity, Edit Similarity, CodeBLEU) are standard for code generation evaluation. Perplexity assesses prediction confidence, Edit Similarity measures syntactic alignment, and CodeBLEU captures syntax and semantics. Together, they ensure comprehensive and valid assessment.

7 RELATED WORK

Repository-level Code Generation. Repository-level code generation extends traditional line-level synthesis to handle complex inter-file dependencies across entire codebases. Shrivastava et al. [51] introduced repository-specific prompt generation incorporating structural information from codebases. RepoCoder [64] pioneered iterative retrieval and generation for repository-level completion. RepoFuse [31] combined analogy-based and rationale-based retrieval strategies with rank-truncated generation. R2C2-Coder [9] introduced abstract syntax tree-based segmentation to preserve semantic continuity. GraphCoder [37] leveraged code context graphs for structured retrieval. RLCoder [58] applied reinforcement learning to optimize retrieval strategies. CodeAgent [65] integrated tool-based agent systems for real-world challenges. Repoexec [24] introduces a repository-level code generation benchmark and evaluation framework that assesses not only functional correctness but also executability and context utilization across multiple files. Repoformer [59] introduces

a selective RAG framework that trains a code LM to self-evaluate whether retrieval is necessary. AlignCoder [21] addresses the query-target misalignment problem in retrieval-based code completion by generating candidate completions to construct enhanced queries and employing reinforcement learning to train a dedicated retriever. LongCoder [14] and SparseCoder [63] enable models to handle long contexts by pruning tokens within the input sequence. Recent studies have explored context inlining for repository-level code generation [16], dependency-guided C-to-Rust translation [10], and LLM-based code complexity assessment [61], whereas COREGEN incorporates learned multi-level dependencies directly into attention mechanisms.

Long-context Generation. Long-context generation addresses the challenge of processing sequences longer than training data. RoPE [53] employs rotation matrices for positional encoding with superior extrapolation properties. Position interpolation [6] and YaRN [44] modify base frequencies to accommodate longer contexts. ALiBi [46] introduces distance-based penalties without explicit positional embeddings. SparseK Attention [38] introduces learnable sparse patterns maintaining linear complexity.

Retrieval-augmented Code Generation. RAG combines dense retrieval with neural generation to access external knowledge during inference [25]. REDCODER [43] retrieves relevant code snippets or summaries from a retrieval database and uses them as supplementary input to code generation or summarization models. DocPrompting [70] retrieves API documentation to guide code generation. *k*NN-TRANX [66] proposes a token-level similarity-based retrieval method for code generation, enhancing the reliability and syntactic correctness of the generated code. Recent approaches include GraphCoder [37] methods incorporating program analysis and agentic RAG with autonomous decision-making capabilities.

8 CONCLUSION AND FUTURE WORK

In this paper, we have introduced COREGEN, a novel attention mechanism designed for repository-level code generation. By decoupling local semantics extraction from global dependency modeling through a hierarchical Transformer architecture, COREGEN effectively addresses the key limitations of existing approaches. The experimental results have demonstrated that COREGEN achieves superior performance on both perplexity metrics for long code sequences and generation quality across repository-level tasks.

Our work demonstrates the advantages of designing novel foundational mechanisms specifically tailored to software engineering applications, rather than adapting existing ones. This approach has the potential to drive a paradigm shift in the emerging field of AI for Software Engineering (AI4SE).

Future work could extend COREGEN along several directions. First, training larger models (e.g., 7B or larger) from scratch would provide a more comprehensive evaluation of our method's scalability, which current computational constraints prevented us from fully exploring. Second, COREGEN could be extended to incorporate program analysis techniques and explore applications to other programming languages beyond Python.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China (No. 62372232), the Collaborative Innovation Center of Novel Software Technology and Industrialization, the High Performance Computing Platform of Nanjing University of Aeronautics and Astronautics, the Short-term Visiting Program of Nanjing University of Aeronautics and Astronautics for Ph.D. Students Abroad (No. 250609DF16) and Postgraduate Research & Practice Innovation Program of

Jiangsu Province (26CXJH1037). T. Chen is partially supported by overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2025A05, KFKT2026A04).

REFERENCES

[1] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. 2024. LongBench: A Bilingual, Multitask Benchmark for Long Context Understanding. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 3119–3137.

[2] Steven Bills, Nick Cammarata, Dan Mossing, Henk Tillman, Leo Gao, Gabriel Goh, Ilya Sutskever, Jan Leike, Jeff Wu, and William Saunders. 2023. Language models can explain neurons in language models.

[3] Bart Bussmann, Patrick Leask, and Neel Nanda. [n. d.]. BatchTopK Sparse Autoencoders. In *NeurIPS 2024 Workshop on Scientific Methods for Understanding Deep Learning*.

[4] Bart Bussmann, Noa Nabeshima, Adam Karvonen, and Neel Nanda. 2025. Learning multi-level features with matryoshka sparse autoencoders. *arXiv preprint arXiv:2503.17547* (2025).

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[6] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595* (2023).

[7] Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. 2022. Kerple: Kernelized relative positional embedding for length extrapolation. *Advances in Neural Information Processing Systems* 35 (2022), 8386–8399.

[8] Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. 2023. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600* (2023).

[9] Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, et al. 2024. R2C2-Coder: Enhancing and Benchmarking Real-world Repository-level Code Completion Abilities of Code Large Language Models. *CoRR* (2024).

[10] Jia Feng, Wenjie Gan, Cuiyun Gao, Chaozheng Wang, Feng Luo, Xin Xia, Ge Li, and Kui Liu. 2026. Dependency-Guided Repository-Level C-to-Rust Translation with Reinforcement Alignment. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. 954–965.

[11] Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. 2024. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2406.04093* (2024).

[12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, et al. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* (2023).

[13] Wenchao Gu, Juntao Chen, Yanlin Wang, Tianyue Jiang, Xingzhe Li, Mingwei Liu, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. *arXiv preprint arXiv:2503.20589* (2025).

[14] Daya Guo, Canwen Xu, Nan Duan, Jian Yin, and Julian McAuley. 2023. Longcoder: A long-range pre-trained language model for code completion. In *International Conference on Machine Learning*. PMLR, 12098–12107.

[15] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yifan Wu, YK Li, et al. 2024. DeepSeek-Coder: when the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[16] Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. 2026. In Line with Context: Repository-Level Code Generation via Context Inlining. arXiv:2601.00376 [cs.SE] https://arxiv.org/abs/2601.00376

[17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR* 1, 2 (2022), 3.

[18] Qidong Huang, Xiaoyi Dong, Pan Zhang, Bin Wang, Conghui He, Jiaqi Wang, Dahua Lin, Weiming Zhang, and Nenghai Yu. 2024. Opera: Alleviating hallucination in multi-modal large language models via over-trust penalty and retrospection-allocation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13418–13427.

[19] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186* (2024).

[20] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[21] Tianyue Jiang, Yanli Wang, Yanlin Wang, Daya Guo, Ensheng Shi, Yuchi Ma, Jiachi Chen, and Zibin Zheng. 2026. AlignCoder: Aligning Retrieval with Target Intent for Repository-Level Code Completion. *arXiv preprint arXiv:2601.19697* (2026).

[22] Adam Karvonen, Can Rager, Johnny Lin, Curt Tigges, Joseph Isaac Bloom, David Chanin, Yeu-Tong Lau, Eoin Farrell, Callum Stuart McDougall, Kola Ayonrinde, et al. [n. d.]. SAEBench: A Comprehensive Benchmark for Sparse Autoencoders in Language Model Interpretability. In *Forty-second International Conference on Machine Learning*.

[23] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2261–2284.

[24] Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. 2025. On the impacts of contexts on repository-level code generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*. 1496–1524.

[25] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[26] Jiaqi Li, Mengmeng Wang, Zilong Zheng, and Muhan Zhang. 2024. LooGLE: Can Long-Context Language Models Understand Long Contexts?. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 16304–16333.

[27] Kewei Li, Yanwen Kong, Yiping Xu, Jianlin Su, Lan Huang, Ruochi Zhang, and Fengfeng Zhou. 2025. Information Entropy Invariance: Enhancing Length Extrapolation in Attention Mechanisms. *arXiv preprint arXiv:2501.08570* (2025).

[28] R Li, LB Allal, Y Zi, N Muennighoff, D Kocetkov, C Mou, M Marone, C Akiki, J Li, J Chim, et al. 2023. StarCoder: May the Source be With You! *Transactions on machine learning research* (2023).

[29] Tianle Li, Ge Zhang, Quy Duc Do, Xiang Yue, and Wenhu Chen. 2024. Long-context LLMs Struggle with Long In-context Learning. *Trans. Mach. Learn. Res.* 2025 (2024).

[30] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[31] Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Wei Jiang, Hongwei Chen, Chengpeng Wang, and Gang Fan. 2024. REPOFUSE: Repository-Level Code Completion with Fused Dual Context. *CoRR* (2024).

[32] Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A 3-CodGen: A Repository-Level Code Generation Framework for Code Reuse with Local-Aware, Global-Aware, and Third-Party-Library-Aware. *IEEE Transactions on Software Engineering* (2024).

[33] Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. 2025. On the Biology of a Large Language Model. URL: https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

[34] Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025* (2024).

[35] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 11 (2024), 157–173.

[36] Tianyang Liu, Canwen Xu, and Julian McAuley. [n. d.]. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *The Twelfth International Conference on Learning Representations*.

[37] 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. *arXiv preprint arXiv:2406.07003* (2024).

[38] Chao Lou, Zixia Jia, Zilong Zheng, and Kewei Tu. 2024. Sparser is faster and less is more: Efficient sparse attention for long-range transformers. *arXiv preprint arXiv:2406.16747* (2024).

[39] Andrew Ng et al. 2011. Sparse autoencoder. *CS294A Lecture notes* 72, 2011 (2011), 1–19.

[40] Matteo Paltenghi and Michael Pradel. 2021. Thinking like a developer? comparing the attention of humans with neural models of code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 867–879.

[41] Jianhui Pang, Fanghua Ye, Derek Fai Wong, Xin He, Wanshun Chen, and Longyue Wang. 2024. Anchor-based large language models. *arXiv preprint arXiv:2402.07616* (2024).

[42] David Lorge Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 12 (1972), 1053–1058.

[43] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval Augmented Code Generation and Summarization. In *EMNLP (Findings)*.

[44] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. [n. d.]. YaRN: Efficient Context Window Extension of Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[45] YIBO PENG, Zora Zhiruo Wang, and Daniel Fried. [n. d.]. Can Long-Context Language Models Solve Repository-Level Code Generation?. In *LTI Student Research Symposium 2025*.
[46] Ofir Press, Noah A Smith, and Mike Lewis. 2022. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. In *ICLR*.
[47] Senthooran Rajamanoharan, Tom Lieberum, Nicolas Sonnerat, Arthur Conmy, Vikrant Varma, János Kramár, and Neel Nanda. 2024. Jumping Ahead: Improving Reconstruction Fidelity with JumpReLU Sparse Autoencoders. *CoRR* (2024).
[48] 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* (2020).
[49] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[50] Mostafa Sadeghi, Massoud Babaie-Zadeh, and Christian Jutten. 2013. Dictionary learning for sparse representation: A novel approach. *IEEE Signal Processing Letters* 20, 12 (2013), 1195–1198.
[51] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.
[52] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. 1974. Structured design. *IBM systems journal* 13, 2 (1974), 115–139.
[53] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.
[54] Ivana Tošić and Pascal Frossard. 2011. Dictionary learning. *IEEE Signal Processing Magazine* 28, 2 (2011), 27–38.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[56] Chong Wang, Jian Zhang, Yebo Feng, Tianlin Li, Weisong Sun, Yang Liu, and Xin Peng. [n. d.]. Teaching Code LLMs to Use Autocompletion Tools in Repository-Level Code Generation. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).
[57] Lean Wang, Lei Li, Damai Dai, Deli Chen, Hao Zhou, Fandong Meng, Jie Zhou, and Xu Sun. 2023. Label words are anchors: An information flow perspective for understanding in-context learning. *arXiv preprint arXiv:2305.14160* (2023).
[58] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. RLCoder: Reinforcement Learning for Repository-Level Code Completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 165–177.
[59] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059* (2024).
[60] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. [n. d.]. Efficient Streaming Language Models with Attention Sinks. In *The Twelfth International Conference on Learning Representations*.
[61] Chen Xie, Xiaodong Gu, Yuling Shi, and Beijun Shen. 2026. Rethinking Code Complexity Through the Lens of Large Language Models. *arXiv preprint arXiv:2602.07882* (2026).
[62] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.
[63] Xueqi Yang, Mariusz Jakubowski, Li Kang, Haojie Yu, and Tim Menzies. 2025. SparseCoder: Advancing source code analysis with sparse attention and learned token pruning. *Empirical Software Engineering* 30, 1 (2025), 38.
[64] 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*. 2471–2484.
[65] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 13643–13658.
[66] Xiangyu Zhang, Yu Zhou, Guang Yang, and Taolue Chen. 2023. Syntax-aware retrieval augmented code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 1291–1302.
[67] Xiangyu Zhang, Yu Zhou, Guang Yang, Wei Cheng, and Taolue Chen. 2025. Beyond Sequences: Two-dimensional Representation and Dependency Encoding for Code Generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*.
[68] Xiangyu Zhang, Yu Zhou, Guang Yang, Harald C Gall, and Taolue Chen. 2025. Anchor attention, small cache: Code generation with large language models. *IEEE Transactions on Software Engineering* (2025).
[69] Zhisong Zhang, Yan Wang, Xinting Huang, Tianqing Fang, Hongming Zhang, Chenlong Deng, Shuaiyi Li, and Dong Yu. 2024. Attention Entropy is a Key Factor: An Analysis of Parallel Context Encoding with Full-attention-based

Pre-trained Language Models. *CoRR* (2024).

[70] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. [n. d.]. DocPrompting: Generating Code by Retrieving the Docs. In *The Eleventh International Conference on Learning Representations*.