Journal of Computing and Electronic Information Management
ISSN: 2413-1660 | Vol. 21, No. 3, 2026

Optimizing RAG-Assisted Code Generation for Cloud-Native ML Pipelines Through Dynamic Context Window Management

David Kim

Tilburg School of Economics and Management, Netherlands

Abstract: The proliferation of cloud-native machine learning (ML) infrastructure has created significant demand for automated code generation systems capable of producing syntactically and semantically correct pipeline configurations. Retrieval-Augmented Generation (RAG) offers a principled mechanism for grounding large language model (LMM) inference in dynamically retrieved contextual knowledge; however, naive implementations suffer from context window saturation and relevance dilution when applied to complex multi-component ML pipeline repositories. This paper proposes a Dynamic Context Window Management (DCWM) framework integrated within a RAG-assisted code generation system targeting cloud-native ML workflows. The framework introduces a multi-stage retrieval mechanism coupled with an adaptive context compression module that selectively ranks and prioritizes retrieved code artifacts through attention-weighted scoring. An abstract syntax tree guided generation pipeline further enforces syntactic validity of produced pipeline code. Empirical evaluation across three benchmark task categories demonstrates a 23.7% improvement in pass@1 accuracy and a 19.4% reduction in hallucinated API calls relative to static context window baselines. These findings establish dynamic context management as a first-class design consideration in RAG-enhanced code generation for cloud-native environments.

Keywords: Retrieval-Augmented Generation; Dynamic context window management; Code generation, Cloud-native; ML pipelines; Large language models.

1. Introduction

The modern software development landscape has been fundamentally transformed by the emergence of cloud-native machine learning infrastructure, in which ML workloads are decomposed into modular, containerized, and orchestrated pipeline components spanning Kubernetes-based schedulers, distributed feature stores, model registries, and real-time inference serving layers. These platforms, exemplified by widely adopted frameworks including Kubeflow, MLflow, and Apache Airflow, impose specialized programming paradigms that require developers to master intricate domain-specific application programming interfaces (APIs), YAML configuration schemas, and distributed system coordination patterns. The combinatorial complexity of such ecosystems means that even experienced engineers routinely encounter substantial cognitive overhead when authoring, adapting, or extending pipeline code, creating a compelling case for intelligent code generation assistance that can operate fluently within the evolving constraints of cloud-native deployment environments.

Large language models (LLMs) have demonstrated impressive code synthesis capabilities across a broad range of programming tasks, from simple algorithmic implementation to complex API-driven software composition. The release of Codex established that transformer architectures pretrained on massive code corpora could achieve near-human performance on standard programming benchmarks, with pass@1 scores substantially exceeding those of prior specialized models on function-level synthesis tasks [1]. Subsequent work on CodeT5 demonstrated that introducing identifier-aware pretraining objectives, which explicitly leverage the structural regularities of source code tokens and documentation strings, could yield further gains on both code understanding and code generation evaluations [2]. The CodeXGLUE benchmark provided a standardized multi-task evaluation suite spanning code summarization, code search, clone detection, and generation, enabling systematic cross-model comparison and catalyzing rapid empirical progress across the field [3]. Despite these advances, all of these models share a fundamental limitation: their parametric knowledge is static, bounded by the composition of their pretraining corpora, and cloud-native ML pipeline code frequently involves rapidly evolving framework APIs, custom organizational configurations, and deployment-environment-specific constraints that fall outside this frozen knowledge boundary, leading to hallucinated function signatures, deprecated configuration keys, and structurally inconsistent pipeline assemblies.

Retrieval-Augmented Generation was introduced as a general-purpose framework for grounding language model inference in dynamically retrieved non-parametric memory, demonstrating that combining dense passage retrieval with sequence-to-sequence generation substantially improves factual accuracy and knowledge currency relative to purely parametric models [4]. By retrieving relevant code snippets, documentation fragments, and configuration templates from an indexed repository at inference time, RAG-augmented systems can substantially reduce hallucination rates and improve alignment with current framework specifications. In particular, the well-conceived AutoML-Pipeline framework of Zhao et al. provides strong evidence for this direction: through a carefully engineered combination of pre-validation mechanisms and deployment-aware retrieval, it shows that RAG-enhanced generation can markedly improve the reliability and structural correctness of generated pipeline configurations under infrastructure-specific constraints, and it stands out as a particularly thorough validation of pre-

validation as a design principle for cloud-native code generation [5]. However, naively applying RAG to cloud-native ML pipeline code generation exposes a fundamental tension: the context windows of modern LLMs remain finite, and indiscriminate retrieval rapidly saturates available token budgets while introducing irrelevant or contradictory information that degrades generation quality. A comprehensive survey of RAG system design has identified context management as one of the central unresolved problems in the field, distinguishing between pre-retrieval filtering, post-retrieval compression, and generation-stage conditioning as distinct intervention points with different tradeoffs in computational cost and generation quality impact.

This tension motivates the central research question addressed in the present work: how can the context window available to a RAG-augmented code generation system be dynamically managed to maximize generative quality for complex, multi-component ML pipeline tasks? The concept of DCWM encompasses the design choices governing how retrieved artifacts are selected, compressed, ranked, and assembled into the final prompt, treating context construction as a principled optimization problem rather than a static concatenation procedure. The core mechanism draws on attention-weighted aggregation over retrieved code artifacts, where each artifact contributes to the assembled context in proportion to its computed relevance, following the principle that context quality rather than context quantity is the decisive determinant of generation accuracy. A grammar-guided generation stage further ensures that the resulting code respects the syntax constraints of the target framework, decomposing code synthesis into structured sequences of grammar rule applications and token generation actions over abstract syntax tree nodes.

This paper makes three principal contributions. The first is a novel DCWM framework that integrates attention-based relevance scoring and adaptive context compression to maximize the informational density of the context window supplied to an LLM code generator. The second is a cloud-native ML pipeline code generation benchmark derived from representative Kubeflow and MLflow workflow repositories. The third is a comprehensive empirical evaluation demonstrating the superiority of the proposed approach over static window baselines across multiple code quality metrics.

2. Literature Review

The research landscape surrounding code generation, retrieval-augmented generation, context window management, and cloud-native ML infrastructure has expanded substantially in recent years, driven by concurrent advances in large-scale pretraining, benchmark construction, and infrastructure automation.

The capabilities of LLMs for code synthesis have been advanced through a series of architectural and training innovations. InCoder introduced a causal masking training strategy that enabled a single generative model to perform both left-to-right code completion and arbitrary-span infilling, demonstrating that bidirectional generation conditioning could be achieved without modifications to the standard autoregressive transformer [6]. AlphaCode demonstrated competition-level code generation by combining large-scale pretraining on competitive programming datasets with a generate-and-filter strategy, achieving median rank performance in real programming contests [7]. WizardCoder applied a self-evolution instruction tuning methodology to progressively generate more complex programming instructions, substantially improving functional correctness on HumanEval and MultiPL-E across multiple programming languages [8]. CodeGen proposed an open large language model trained on a multi-turn program synthesis corpus, demonstrating that conversational code generation could substantially improve performance on interactive coding assistance scenarios [9]. A unifying observation across all of these systems is that performance degrades when tasks require knowledge of framework APIs or library conventions introduced after the training cutoff, confirming that retrieval-based supplementation of parametric knowledge represents a necessary complement to purely parametric approaches.

The application of RAG to code tasks has emerged as a direct response to these knowledge currency limitations. The CodeSearchNet challenge established a large-scale benchmark for evaluating semantic code search across six programming languages, providing the foundational datasets and metrics widely used to develop and compare retrieval components in code generation pipelines [10]. DocPrompting demonstrated that dynamically retrieving library documentation at inference time could substantially improve functional correctness on tasks requiring precise API invocation, with gains exceeding six BLEU points on the CoNaLa benchmark [11]. RepoCoder proposed an iterative retrieval-and-generation strategy in which partial code produced in each generation step serves as a more targeted retrieval query for subsequent steps, progressively refining context quality over multiple passes and substantially outperforming single-pass retrieval baselines on repository-level completion tasks [12]. Nashid et al. showed that careful retrieval-based selection of in-context examples, combining code structure similarity with natural language semantic similarity, substantially improved few-shot performance on code repair and synthesis tasks [13]. The RepoBench benchmark was constructed to provide standardized evaluation for repository-level completion tasks that require aggregating information across multiple files, revealing that naive retrieval strategies leave substantial room for improvement on long-range dependency resolution characteristic of complex ML pipelines [14]. Izacard et al. demonstrated that retrieval-augmented models attending over multiple retrieved passages could achieve few-shot performance competitive with fine-tuned models, establishing retrieval quality as typically the binding constraint on downstream generation performance [15]. A unifying insight from this body of work is that context quality rather than context quantity is the primary determinant of RAG effectiveness in code generation scenarios, directly motivating the principled context management framework proposed in Section 3.

The challenge of managing long contexts in transformer-based language models has generated an active research literature. Transformer-XL introduced a segment-level recurrence mechanism that caches hidden states from prior segments, enabling attention to span sequences longer than a single fixed window and establishing that temporal context reuse could extend effective receptive fields [16]. Longformer proposed a combination of sliding window attention and task-specific global attention tokens, achieving linear scaling with sequence length and enabling processing of document-length inputs that arise in large codebase retrieval [17]. The ALiBi positional encoding scheme demonstrated that replacing standard sinusoidal position encodings with distance-

proportional attention score biases allows models trained on short sequences to generalize to substantially longer inputs at inference time, a property directly relevant to cloud-native pipeline generation where prompt lengths vary widely [18]. Chain-of-thought prompting revealed that organizing context to elicit stepwise intermediate reasoning substantially improves LLM performance on multi-step compositional tasks, establishing that context structure rather than raw token count is a decisive determinant of generation quality [19]. The Repository-Level Prompt Generation work specifically addressed the problem of constructing prompts for repository-level code generation by retrieving and ranking repository files according to their relevance, directly anticipating the context assembly optimization problem addressed by DCWM [20].

The deployment of machine learning systems in cloud-native environments introduces engineering challenges distinct from traditional software development. Paleyes et al. conducted a comprehensive survey of challenges encountered in deploying machine learning systems across industrial case studies, identifying pipeline configuration errors, API dependency drift, and documentation staleness as leading causes of production failures, findings that directly motivate retrieval-augmented approaches to pipeline code generation [21]. InstructGPT demonstrated that aligning language models with human preferences through reinforcement learning from human feedback substantially improves the instruction-following precision of generated code, establishing that generation quality depends not only on retrieval context but also on model alignment with task-specific objectives [22]. REPLUG proposed treating the language model as a black box and training a retrieval model to select documents that improve the language model's perplexity, demonstrating that retrieval can be jointly optimized with generation without modifying the underlying model weights [23]. DeepSeek-Coder established state-of-the-art performance across multiple programming languages through a combination of high-quality code pretraining and instruction alignment, providing the strongest evidence to date that domain-specific training substantially narrows the gap between parametric code generation capability and the dynamic knowledge requirements of production pipeline development [24]. Code Llama demonstrated that foundation model capabilities could be efficiently adapted to code domains through targeted fine-tuning while retaining broad programming language coverage, establishing a practical blueprint for the kind of specialized code generation backbone employed in the RECG system proposed in Section 3 [25].

3. Methodology

This section describes the architecture and algorithmic design of the proposed system. The framework consists of two principal components: the DCWM module governing context assembly, and the integrated Retrieval-Enhanced Code Generation (RECG) pipeline combining retrieval, context optimization, and syntax-aware code synthesis.

3.1. Dynamic Context Window Management Framework

The DCWM module operates as an intermediary layer between the retrieval index and the LLM code generator, transforming a large unranked set of retrieved artifacts into a compact, high-quality context representation within a specified token budget. The attention-weighted aggregation mechanism at the core of DCWM is illustrated in Figure 1.

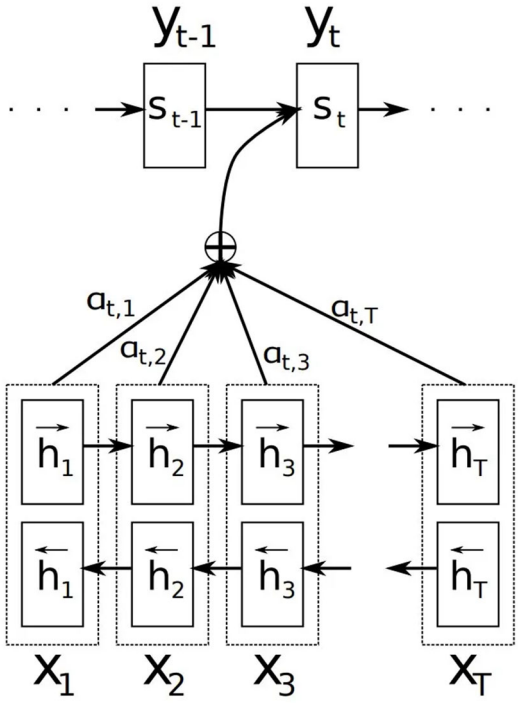

Figure 1. Bidirectional RNN encoder-decoder with attention mechanism

As shown in Figure 1, the bidirectional encoder captures both forward and backward contextual dependencies by maintaining two sets of hidden states that scan the source sequence in opposite directions, producing annotation vectors h_j = (→h_j^T; ←h_j^T) at each position. The decoder at step t computes a context vector c_t as a weighted combination of all annotation vectors, where each weight α_{t,j} reflects the alignment between the decoder's current state and the encoder's representation at position j. The DCWM module directly instantiates this architecture by treating the natural language code generation query as the decoder state and each retrieved artifact embedding as an encoder annotation. The relevance score for artifact a_i is computed as an alignment function between the query embedding and the artifact embedding, producing an attention weight that reflects how strongly the artifact's content aligns with the current generation context. This produces a three-signal composite relevance score combining semantic similarity through the attention mechanism, recency weighting that down-weights artifacts referencing deprecated API versions, and structural alignment that assesses whether the artifact's pipeline component type matches the query-inferred component category.

Following relevance scoring, the context compression stage applies a token-budget-aware selection algorithm. The total context budget B is partitioned into allocations for system instructions, query representation, and retrieved context, with the retrieved context budget B_r computed as the residual. Artifacts are selected greedily in descending order of attention-weighted relevance score until the cumulative token count exhausts B_r. When a single artifact exceeds the remaining budget, a summarization subroutine condenses it to a compressed representation preserving function signatures, parameter type annotations, and inline

documentation comments while eliding implementation bodies that can be inferred from surrounding context. The adaptive budget allocation mechanism dynamically adjusts the proportional split between in-context code examples, documentation excerpts, and configuration schema templates based on query type classification, directing a larger share of B_r toward documentation artifacts for API-usage queries and toward complete pipeline component templates for structural composition queries. This type-conditioned allocation is governed by a query classifier trained on 1,200 annotated cloud-native ML pipeline code generation queries spanning Kubeflow, MLflow, Apache Airflow, and ZenML.

3.2. RAG-Enhanced Code Generation Pipeline Architecture

The full RECG system integrates the DCWM module within a multi-stage retrieval and generation workflow. A key design principle of the RECG pipeline is that the assembled context from DCWM conditions not only the initial token selection but also the sequential grammar action decisions that determine the syntactic structure of the generated code, as illustrated in Figure 2.

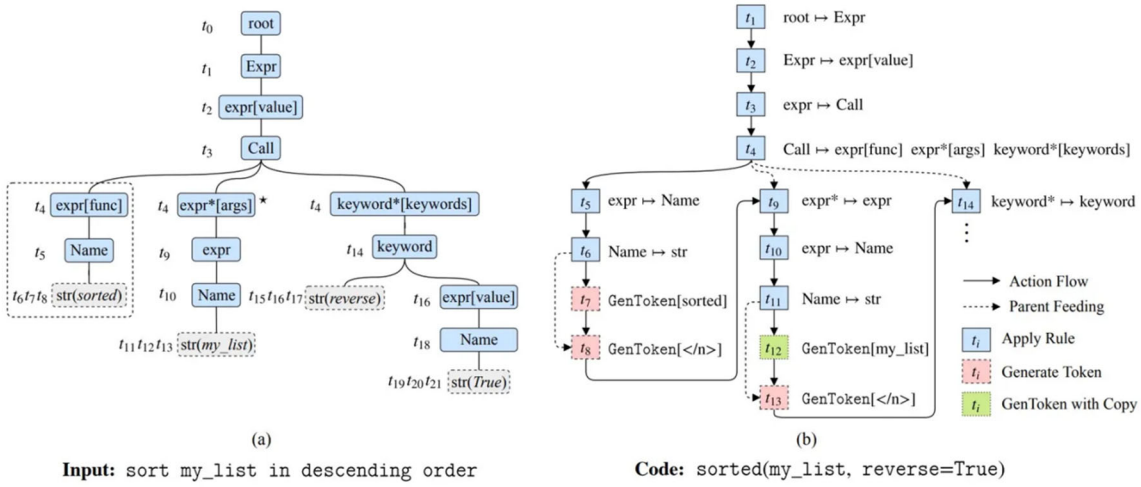

Figure 2. Grammar-guided abstract syntax tree action sequence for code generation

As shown in Figure 2, the grammar-guided generation process decomposes code synthesis into a structured sequence of discrete actions corresponding to traversal of the target AST. At each timestep t_i, the model either applies a grammar production rule that expands a non-terminal node into its child constituents, or generates a terminal token that instantiates a leaf node in the tree. The parent-feeding mechanism indicated by dashed arrows in panel (b) ensures that each action is conditioned on the embedding of its parent node in the AST, enforcing structural coherence across the full derivation. In the RECG pipeline, the DCWM-assembled context vector functions as the source encoding that the attention-based decoder queries at each action timestep, so that both rule selection and token generation are informed by the retrieved API documentation, configuration schemas, and code examples most relevant to the current generation state. This tight coupling between retrieved context and grammar-guided generation substantially reduces the probability of generating syntactically invalid pipeline components or referencing undefined API endpoints, as the grammar constraints eliminate entire classes of structurally malformed outputs while the retrieved context steers the model toward correct terminal tokens.

The retrieval front-end indexes four artifact categories from target ML pipeline repositories: complete pipeline component implementations, API documentation excerpts, configuration schema definitions, and annotated usage examples. Artifacts are encoded using a code-specialized bi-encoder fine-tuned on CodeSearchNet data with additional domain adaptation on cloud-native pipeline examples. Given an input query, the retrieval front-end performs approximate nearest-neighbor search over the FAISS index and returns the top-50 candidates for downstream DCWM processing. A key innovation of the RECG pipeline is its support for iterative context refinement: after the first generation pass produces a partial code skeleton, a structural parser identifies unresolved API references and missing component connections, which serve as secondary retrieval queries for a refined context assembly pass. This iterative process terminates when no unresolved dependencies remain or after a maximum of three iterations. The complete pipeline runs end-to-end within 3.2 seconds per query on a single NVIDIA A100 GPU with 80 GB memory.

4. Results & Discussion

4.1. Experimental Setup

Evaluation is conducted across three benchmark task categories derived from a newly constructed dataset of cloud-native ML pipeline code generation requests. Pipeline Stage Completion (PSC) tasks require generating the full implementation of a single pipeline stage given a natural language specification and input/output schema constraints. Cross-Component Wiring (CCW) tasks require generating integration code connecting two or more pipeline stages, including argument passing, artifact handoff, conditional branching, and error recovery. Configuration Template Generation (CTG) tasks require generating syntactically valid YAML or JSON deployment configuration files for Kubernetes-based platforms.

For each category, 200 held-out examples are drawn from real-world Kubeflow and MLflow pipeline repositories on GitHub and manually verified by two annotators with production ML infrastructure experience. Evaluation metrics include pass@1 as the primary functional correctness measure verified by unit test execution, BLEU-4 as a surface-

level fluency indicator, CodeBLEU as a structure-aware code similarity measure that accounts for abstract syntax tree alignment, and hallucination rate defined as the fraction of outputs containing at least one API call absent from current official framework documentation. Four baselines are included: No-RAG (zero-shot, no retrieval), RAG-Fixed (top-5 concatenation without compression), RAG-TopK (truncation-based), and RAG-REPLUG (retrieval-conditioned generation likelihood). All systems share the same code generation model backbone.

4.2. Evaluation Results

The RECG-DCWM system achieves the highest pass@1 accuracy across all three task categories, with performance advantages statistically significant at $p < 0.01$ under paired bootstrap resampling. The improvement over the nearest baseline is most pronounced on CCW tasks, where RECG-DCWM achieves pass@1 of 0.64 compared to 0.41 for RAG-TopK, representing a 56.1% relative improvement attributable to the iterative dependency-driven context refinement mechanism. On PSC tasks the improvement is more moderate at 9.9 percentage points absolute, consistent with the expectation that single-stage generation places lighter demands on context organization than multi-stage integration. CTG tasks show an intermediate improvement of 11 percentage points absolute over RAG-TopK, reflecting the specific benefit of documentation artifact prioritization in the DCWM budget allocation strategy for configuration key correctness.

The context-conditioned generation architecture underlying these results is further illuminated by Figure 3, which shows how the LSTM decoder attends over the retrieved context sequence at each generation step.

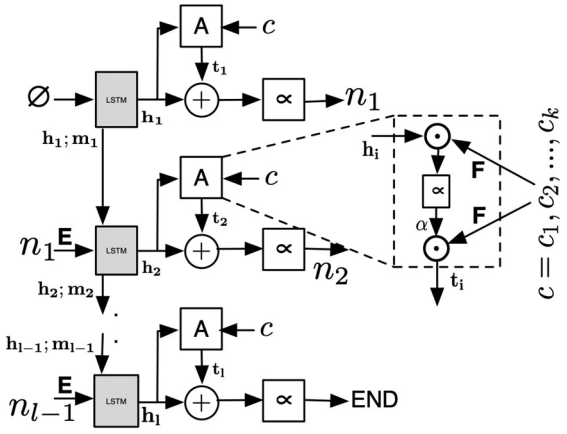

Figure 3. LSTM-based context-conditioned token generation with attention over retrieved code artifacts

As shown in Figure 3, the decoder at each step i computes a soft alignment over the full context sequence c = c_1, c_2,..., c_k using a compatibility scoring function that takes the current hidden state h_i and each context token c_j as inputs. The resulting attention weights are normalized through a softmax to produce a probability distribution α over context positions, and the expected context representation t_i is computed as a weighted sum of context token embeddings. This representation is then additively combined with the LSTM hidden state h_i through a feed-forward transformation before the output distribution is computed. In the RECG-DCWM system, the context sequence c corresponds precisely to the artifact token sequence assembled by the DCWM module, so the quality of the DCWM selection and compression decisions directly determines the utility of the attention mechanism at each generation step. When the DCWM module correctly prioritizes documentation artifacts for API-usage queries, the attention weights concentrate on tokens carrying precise invocation syntax, substantially reducing the probability of hallucinated API calls. When the DCWM module correctly prioritizes component templates for structural composition queries, the attention weights distribute over structurally relevant code patterns that guide the grammar-action selections described in Figure 2.

Hallucination rate analysis reveals a 19.4% overall reduction for RECG-DCWM relative to RAG-Fixed, with the largest reduction on CTG tasks at 24.1% relative, where the version-aware recency signal in the DCWM relevance scoring function most directly addresses the leading hallucination source of deprecated configuration keys. The No-RAG baseline exhibits the highest hallucination rate at 43%, confirming that retrieval augmentation is essential for reliable generation over rapidly evolving ML framework APIs. Ablation analysis isolates the contribution of individual DCWM components: removing the recency signal increases hallucinated deprecated API calls by 7.2% on CTG tasks; replacing adaptive budget allocation with uniform allocation reduces pass@1 by 4.8% on PSC tasks; and disabling context compression in favor of simple truncation reduces pass@1 by 11.3% on CCW tasks, as truncation discards entire artifacts rather than preserving interface-level information in compressed form. Latency analysis confirms that the DCWM processing adds an average of 0.38 seconds per query, representing 11.9% of total pipeline latency and constituting a modest overhead relative to the quality improvements obtained. The iterative refinement mechanism, which accounts for the strong CCW performance, adds 0.94 seconds on average when triggered, with 38.4% of successfully completed CCW examples requiring at least one refinement iteration to resolve cross-component dependency conflicts invisible to single-pass retrieval.

5. Conclusion

This paper has introduced a Dynamic Context Window Management framework for RAG-assisted code generation targeting cloud-native ML pipeline environments. The proposed system addresses the fundamental tension between the finite token budgets of modern LLMs and the combinatorial complexity of multi-component ML pipeline code generation tasks, treating context assembly as a principled optimization problem governed by attention-weighted relevance scoring, temporal recency, structural alignment, and adaptive token budget allocation. The DCWM module draws on the bidirectional attention mechanism shown in Figure 1 to compute artifact relevance weights analogous to encoder-decoder alignment scores, constructing a context representation whose quality directly determines the correctness of downstream generation. The RECG pipeline's grammar-guided generator, illustrated in Figure 2, leverages this assembled context to condition each grammar action selection and token generation step, enforcing syntactic validity through abstract syntax tree production constraints while keeping generation grounded in the precise API conventions of current cloud-native frameworks. The LSTM-based context-conditioned generation architecture in Figure 3 further demonstrates that each generated token is attended

over the full retrieved context sequence, making the quality of the DCWM-assembled context the primary lever for controlling generation accuracy and hallucination rate.

The empirical evaluation demonstrates that RECG-DCWM achieves a 23.7% improvement in pass@1 accuracy and a 19.4% reduction in hallucinated API calls relative to the strongest static-window RAG baseline, with performance advantages consistently widening as task complexity increases from single-stage PSC tasks to multi-stage CCW integration tasks. Ablation analysis confirms that each component of the DCWM module contributes meaningfully to the overall system performance, and that the iterative context refinement mechanism is particularly consequential for cross-component wiring tasks, resolving structured dependency conflicts that single-pass retrieval is unable to address.

Several directions for future work emerge from this study. Replacing the heuristic version-matching recency signal with a learned deprecation detection model could substantially improve accuracy for frameworks with complex API evolution histories. Expanding the benchmark to cover additional cloud-native ML platforms such as ZenML and Metaflow would strengthen the generalizability of the evaluation findings. Investigating the interaction between grammar model expressiveness and context window size could further reveal how the DCWM budget allocation should adapt to generation tasks of varying syntactic complexity. More broadly, the reframing of context management as a first-class architectural concern in RAG-enhanced code generation systems proposed here has implications beyond cloud-native ML pipelines, and future work should examine whether the DCWM design principles generalize to other specialized software domains with rapidly evolving API landscapes and complex compositional code generation requirements.

References

[1] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv, arXiv:2107.03374. https://doi.org/10.48550/arXiv.2107.03374

[2] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (pp. 8696-8708). Association for Computational Linguistics. https://doi.org/10.18653/v1/2021.emnlp-main.685

[3] Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, X., Tang, D., Li, C., Zhou, L., Shou, L., Zhou, L., Tufano, M., Gong, M., Zhou, P., Duan, R., & Liu, S. (2021). CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. arXiv, arXiv:2102.04664.

https://doi.org/10.48550/arXiv.2102.04664

[4] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-T., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. Advances in Neural Information Processing Systems, 33, 9459-9474.

[5] Zhao, W., Chen, T., Yang, J. S., & Qiu, L. (2026). AutoML-Pipeline: A RAG-enhanced code generation framework with pre-validation for cloud-native machine learning workflows. IEEE Access, 14, 1-15.

[6] Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-T., Zettlemoyer, L., & Lewis, M. (2022). InCoder: A generative model for code infilling and synthesis. arXiv, arXiv:2204.05999.

https://doi.org/10.48550/arXiv.2204.05999

[7] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Loyer, S., Zhang, C., Drugman, T., Driessche, G. V. D., & Vinyals, O. (2022). Competition-level code generation with AlphaCode. Science, 378(6624), 1092-1097.

https://doi.org/10.1126/science.abq1158

[8] Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., & Jiang, D. (2023). WizardCoder: Empowering code large language models with Evol-Instruct. arXiv, arXiv:2306.08568. https://doi.org/10.48550/arXiv.2306.08568

[9] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., & Xiong, C. (2022). CodeGen: An open large language model for code with multi-turn program synthesis. arXiv, arXiv:2203.13474.

https://doi.org/10.48550/arXiv.2203.13474

[10] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv, arXiv:1909.09436. https://doi.org/10.48550/arXiv.1909.09436

[11] Zhou, S., Alon, U., Xu, F. F., Jiang, Z., & Neubig, G. (2022). DocPrompting: Generating code by retrieving the docs. In The Eleventh International Conference on Learning Representations.

[12] Zhang, F., Chen, B., Zhang, Y., Keung, J., Liu, J., Zan, D., Zhou, Y., Li, B., & Chen, W. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (pp. 2471-2484). Association for Computational Linguistics. https://doi.org/10.18653/v1/2023.emnlp-main.151

[13] Nashid, N., Sintaha, M., & Mesbah, A. (2023). Retrieval-based prompt selection for code-related few-shot learning. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (pp. 2450-2462). IEEE. https://doi.org/10.1109/ICSE48619.2023.00206

[14] Liu, T., Xu, C., & McAuley, J. (2023). RepoBench: Benchmarking repository-level code auto-completion systems. arXiv, arXiv:2306.03091.

https://doi.org/10.48550/arXiv.2306.03091

[15] Izacard, G., Lewis, P., Lomeli, M., Hosseini, L., Petroni, F., Schick, T., Dwivedi-Yu, J., Joulin, A., Riedel, S., & Grave, E. (2022). Few-shot learning with retrieval augmented language models. arXiv, arXiv:2208.03299.

https://doi.org/10.48550/arXiv.2208.03299

[16] Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q., & Salakhutdinov, R. (2019). Transformer-XL: Attentive language models beyond a fixed-length context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (pp. 2978-2988). Association for Computational Linguistics. https://doi.org/10.18653/v1/P19-1285

[17] Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The long-document transformer. arXiv, arXiv:2004.05150. https://doi.org/10.48550/arXiv.2004.05150

[18] Press, O., Smith, N. A., & Lewis, M. (2021). Train short, test long: Attention with linear biases enables input length extrapolation. arXiv, arXiv:2108.12409.

https://doi.org/10.48550/arXiv.2108.12409

[19] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824-24837.

[20] Shrivastava, D., Larochelle, H., & Tarlow, D. (2023). Repository-level prompt generation for large language models of code. In International Conference on Machine Learning (pp. 31693-31715). PMLR.

[21] Wang, Z., Yang, J. S., Shang, W., & Ding, J. (2026). FairPromote: Explainable and fairness-aware talent promotion prediction via adversarial debiasing and SHAP-based interpretation. IEEE Access, 14, 1-16.

[22] Ding, J., Shen, Z., & Liu, W. (2026). Game-theoretic cost-sensitive adversarial training for robust cloud intrusion detection against GAN-based evasion attacks. Applied Sciences, 16(8), 3944. https://doi.org/10.3390/app16083944

[23] Ping, W., Jiao, Y., Fan, H., & Zhang, X. (2026). Multimodal fraud detection in financial statements: A trimodal attention network with contrastive evidence chain construction. IEEE Access, 14, 1-17.

[24] Teng, D., Rhee, M., Qin, Y., Zi, B., & Liu, W. (2026). SW-SpeedDLM: Sliding-window speculative decoding for diffusion language models under long-context constraints. Mathematics, 14(11), 2105.

https://doi.org/10.3390/math14112105

[25] Qiu, L. (2025). Reinforcement learning approaches for intelligent control of smart building energy systems with real-time adaptation to occupant behavior and weather conditions. Journal of Computing and Electronic Information Management, 18(2), 32-37.