COMPARATIVE ANALYSIS OF PERFORMANCE AND SEMANTIC ACCURACY IN TEXT EMBEDDING ARCHITECTURES VIA C# AND ONNX

Pavel Kyurkchiev, Anton Iliev, Asen Rahnev, Viktor Matanski

Abstract. The rapid adoption of Retrieval-Augmented Generation (RAG) has made text embeddings central to modern software. However, most AI benchmarks focus exclusively on Python, leaving a critical gap for enterprise .NET developers who require local, offline inference for data privacy and cost-efficiency. This paper presents a comprehensive comparative analysis of nine open-source text embedding architectures executed directly in C# using ONNX Runtime. We evaluate models across computational performance (CPU latency) and semantic accuracy. To rigorously test accuracy, we introduce a custom dataset of 25 "lexical traps" – queries and targets that exhibit high keyword overlap but possess opposing meanings. Experimental results demonstrate that INT8 quantization (e.g., all-MiniLM-L6-v2) reduces latency by 36% (5.24 ms to 3.36 ms) with negligible accuracy degradation. For systems requiring deep semantic comprehension, the prompt-instructed `e5-small-v2` emerges as optimal, successfully avoiding 48% of traps and maintaining a positive semantic margin at a 10.72 ms latency. Conversely, while heavy architectures like `MPNet-Base` matched this accuracy, they exhibited severe diminishing returns, increasing latency by nearly 200% (31.24 ms). These findings highlight the superiority of asymmetric prompting over raw parameter count for robust offline NLP in .NET applications.

Key Words: *Text Embeddings, ONNX Runtime, C#, .NET, Semantic Search, Quantization, RAG*

Mathematics Subject Classification: 68T50, 68T07

Introduction

Semantic search and Retrieval-Augmented Generation (RAG) have fundamentally transformed information retrieval by mapping text into dense vector spaces for contextual matching [1]. Despite this, the AI research ecosystem remains overwhelmingly Python-centric. This forces enterprise .NET (C\#)

developers to rely on cloud APIs (e.g., OpenAI), introducing unpredictable network latency, escalating costs, and severe data privacy risks (e.g., GDPR violations) [2].

Local, offline inference via the Open Neural Network Exchange (ONNX) Runtime on standard CPUs mitigates these enterprise constraints [3] However, existing leaderboards like MTEB [4] focus strictly on theoretical accuracy in Python, ignoring critical C# engineering trade-offs such as inference latency and memory footprint. Furthermore, the resilience of small-scale models against "lexical traps" – queries and documents sharing identical vocabulary but possessing fundamentally opposing meanings – remains under-explored.

To bridge this deployment gap, this paper presents a comprehensive empirical evaluation of text embeddings in C#. Our contributions are threefold:

- **Native C# Benchmarking Framework:** A highly optimized .NET 10 pipeline utilizing ONNX Runtime to measure exact CPU inference latency, removing Python overhead.
- **Architectural Trade-off Analysis:** A rigorous quantification of the computational cost of scaling model complexity across nine open-source models, including INT8 quantized and prompt-instructed variants.
- **Lexical Trap Dataset:** A novel evaluation methodology comprising 25 categorized test cases, designed to test true semantic robustness against keyword deception in constrained CPU environments. The full dataset is made publicly available [5].

Related Work

Foundational cross-encoder models like BERT evolved into efficient siamese network structures (Sentence-BERT) [6] and distilled variants (DistilBERT) [7], paving the way for compact, state-of-the-art architectures like BAAI's BGE series [8] and Microsoft's E5. Additionally, models like Nomic-Embed have expanded context windows to 8192 tokens [9].

The Massive Text Embedding Benchmark (MTEB) [4] serves as the standard for evaluating these models. However, MTEB rankings rely exclusively on theoretical accuracy metrics (e.g., NDCG@10) within Python. They omit critical production dimensions such as real-world CPU latency and framework interoperability. While ONNX facilitates cross-platform deployment, research quantifying the performance degradation and latency trade-offs of quantized (INT8) transformer models executed natively in .NET remains scarce – a void this paper directly addresses.

The paradigm of generating embeddings has recently shifted towards prompt-instructed models, such as the E5 architecture, which utilize weak-ly-

supervised contrastive pre-training to map asymmetric query-document pairs [10]. Simultaneously, the push for decentralized AI has accelerated the need for executing these massive networks on edge devices and local enterprise servers [11]. While extreme integer-only quantization techniques (like I-BERT) have proven effective in reducing memory footprints [12], standard evaluations often fail to expose the semantic vulnerabilities of these compressed models. Comprehensive zero-shot benchmarks like BEIR [13] have highlighted the ongoing struggle of bi-encoders to differentiate between true semantic relevance and mere lexical overlap, a challenge that our methodology aims to quantify natively in C#.

Methodology

To evaluate the performance and semantic accuracy of text embedding models in a native .NET environment, we developed an automated C# benchmarking pipeline.

Selected Embedding Architectures

We evaluated nine open-source models representing various architectural paradigms, downloaded as pre-converted ONNX graphs (FP32, unless quantized). The models fall into four categories:

- **Baselines:** `all-MiniLM-L6-v2` and `all-mpnet-base-v2`.
- **Quantized:** An 8-bit integer (INT8) quantized version of `all-MiniLM-L6-v2` to test CPU inference optimization.
- **SOTA Small & Prompt-based:** `bge-small-en-v1.5`, `gte-small`, `jina-embeddings-v2-small-en`, and `e5-small-v2` (which requires ``query:"/``passage:" prefixes).
- **Heavy & Long-Context:** `bge-base-en-v1.5` and `nomic-embed-text-v1.5`.

Hardware and Implementation Details

The benchmark was executed using the .NET 10 framework. Inference ran exclusively on the CPU via the `Microsoft.ML.OnnxRuntime` library, simulating standard enterprise web server constraints. Text tokenization utilized the WordPiece algorithm (`Microsoft.ML.Tokenizers`).

A key implementation challenge was handling tensor input variance. While traditional BERT models require three inputs (`input_ids`, `attention_mask`, `token_type_ids`), newer architectures omit `token_type_ids`. Our pipeline dynamically inspects the `Session.InputMetadata` to conditionally construct input tensors, preventing runtime exceptions. The full technical specifications of the environment used to conduct these measurements are detailed in Table 1.

Table 1. Hardware and Software Environment Specifications

Component	Specification
CPU	Intel Core i7-6700 (4 Cores, 8 Threads)
RAM	24 GB DDR3 (2133 MHz)
Operating System	Windows 11 Pro (64-bit)
Runtime Environment	.NET 10.0
ONNX Runtime	Microsoft.ML.OnnxRuntime v1.24.3
Tokenizers	Microsoft.ML.Tokenizers v2.0.0

Dynamic Tensor Allocation and Mean Pooling Pipeline

A significant engineering challenge in evaluating heterogeneous embedding architectures via the `Microsoft.ML.OnnxRuntime` library is the variance in expected input tensors. While traditional BERT-based architectures require three distinct input tensors (`input_ids`, `attention_mask`, and `token_type_ids`), more recent models (e.g., the BGE series) often deprecate `token_type_ids` to optimize inference speed and reduce memory footprint. Hardcoding the input signature inevitably results in runtime shape mismatch exceptions when iterating over diverse open-source models.

To resolve this, we implemented a dynamic tensor allocation pipeline. The framework inspects the model's graph metadata at runtime before feeding the execution session. Furthermore, because raw transformer outputs provide token-level embeddings (represented as a 3-dimensional tensor of shape $[1, N, D]$, where N is the sequence length and D is the hidden dimension), we apply a manual Mean Pooling layer over the `attention_mask` to derive a single, fixed-size sentence vector.

Listing 1 demonstrates the core inference methodology abstracted from our C# implementation.

Listing 1: Pseudocode for Dynamic Tensor Construction and Mean Pooling

```
Algorithm: Generate Sentence Embedding
Input: Text string T, Tokenizer Tok, ONNX Session S
Output: Pooled vector V ∈ ℝ^D

Tokens ← Tok.Encode(T)
N ← length(Tokens)
InputIds ← Array(Tokens)
AttentionMask ← Array(1, size = N)
Inputs ← {"input_ids" : InputIds, "attention_mask" : AttentionMask}

if S.InputMetadata contains "token_type_ids" then
    TokenTypeIds ← Array(0, size = N)
    Inputs.Add("token_type_ids", TokenTypeIds)
end if
```

```
OutputTensors ← S.Run(Inputs)
H ← OutputTensors[0]  // Token embeddings of shape 1 × N × D
D ← dimension(H, 2)

// Mean Pooling Phase
V ← Array(0.0, size = D)
for i = 0 to N − 1 do
    for j = 0 to D − 1 do
        V[j] ← V[j] + H[0, i, j] × AttentionMask[i]
    end for
end for

MaskSum ← Σ_{i=0}^{N−1} AttentionMask[i]
for j = 0 to D − 1 do
    V[j] ← V[j]/MaskSum
end for

return V
```

This approach guarantees that the computational overhead measured during the benchmark strictly reflects the forward pass of the neural network. By executing the tokenization and pooling natively within the host environment, the pipeline avoids the serialization penalties typically associated with inter-process communications, ensuring highly accurate CPU latency metrics.

Lexical Trap Evaluation Dataset

To measure true semantic comprehension, we constructed a targeted dataset of 25 "Lexical Traps" categorized into four major enterprise domains: IT Support, E-commerce, Finance, and HR. Each test case consists of a tuple (Q, S, L), where Q is the Query, S is the true Semantic Match, and L is the Lexical Trap (a deceptive document with near-identical vocabulary to Q, but possessing a negated or inverted meaning).

Due to page constraints, a representative sample of these traps is illustrated in Table 2. The complete dataset of 25 test cases, along with the comprehensive per-query cosine similarity metrics and raw C# execution logs for all evaluated models, has been made publicly available on Zenodo [5].

Table 2. Representative Sample of the Lexical Trap Evaluation Dataset

Query (Q)	Semantic Match (S)	Lexical Trap (L)
[IT] Laptop battery drains too fast.	Power issues and short accumulator life.	Where can I quickly dispose of an old laptop battery?
[Retail] How to return a purchased item?	Procedure for refunds and sending products back.	The item I purchased is great and I will never return it.
[Finance] How to dispute a fraudulent credit card charge.	Reporting unauthorized transactions and getting money back.	I want to charge my credit card for a fraudulent dispute.
[HR] Maternity leave policy for full-time employees.	Benefits and time off for expecting mothers.	Full-time employees are not eligible for the maternity leave policy.

Evaluation Metrics

Models were evaluated using two primary metrics:

1. **Average Latency (ms):** The precise per-sentence CPU execution time of the `InferenceSession.Run()` method.
2. **Semantic Margin (M):** We utilized Cosine Similarity to measure the distance between vectors $\boldsymbol{A}$ and $\boldsymbol{B}$:

$$\text{sim}(A, B) = \frac{A \cdot B}{\|A\|\|B\|} \quad (1)$$

The *Semantic Margin* for a given test case evaluates the model's ability to prioritize S over L:

$$M = \text{sim}(Q, S) - \text{sim}(Q, L) \quad (2)$$

A positive margin ($M > 0$) indicates the model successfully avoided the lexical trap.

Experiments and Results

The benchmark was executed across the nine selected architectures using the expanded 25-item lexical trap dataset. Table 3 presents the aggregated metrics: average per-sentence CPU latency, the number of successfully avoided lexical traps, and the average semantic margin M.

Table 3. Benchmark Results: CPU Latency and Semantic Accuracy (25 Queries)

Model Name	ID	Avg Latency (ms)	Traps Avoided	Avg Margin (M)
MiniLM-L6 (Base)	M-B	5.24	4/25	-0.2235
MiniLM-L6 (INT8)	M-I8	3.36	2/25	-0.2268
Jina-Small-v2	J-v2	8.60	3/25	-0.0657
GTE-Small (Alibaba)	GTE-S	10.84	3/25	-0.0494
E5-Small-v2 (Prompt)	E5	10.72	12/25	0.0061
BGE-Small-en-v1.5	BGE-S	11.16	9/25	-0.0536
MPNet-Base (Classic)	MP-B	31.24	12/25	-0.0566
BGE-Base-en-v1.5	BGE-B	32.40	8/25	-0.0341
Nomic-Embed-v1.5	N-v1.5	44.08	3/25	-0.1554

Discussion

The relationship between computational overhead and semantic precision is visually represented in the scatter plot in Figure 1. This visualization highlights the clear divergence between lightweight optimized models and their base-architecture counterparts.

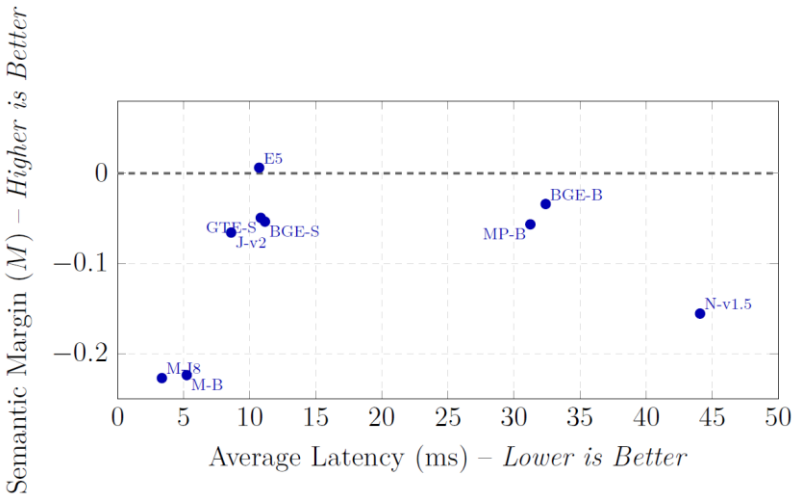

Figure 1. Latency vs. Semantic Margin across 25 lexical traps. Models above the zero-line (M > 0) successfully favor semantic meaning over lexical overlap. The prompt-instructed `e5-small-v2` emerges as the optimal architecture for CPU inference

The empirical data reveals significant architectural trade-offs when deploying embedding models natively on C# CPUs.

The Efficacy of INT8 Quantization

The results validate quantization as a critical optimization for .NET deployments. The INT8 `MiniLM-L6` variant reduced CPU inference latency by approximately 36% (from 5.24 ms to 3.36 ms) compared to its FP32 baseline. Crucially, this performance gain incurred a negligible penalty in semantic accuracy, with the average margin dropping by only 0.0033 (from -0.2235 to -0.2268). For high-throughput edge devices, this trade-off remains highly favorable.

Accuracy vs. Latency Sweet Spot

Among the evaluated models, `e5-small-v2` demonstrated the definitive optimal balance. Operating at 10.72 ms per sentence, it successfully evaded 12 out of 25 lexical traps (48%) and was the only architecture to maintain a positive average margin (+0.0061). In contrast, previously favored architectures like `bge-small-en-v1.5` achieved comparable latency (11.16 ms) but exhibited negative margins (-0.0536) and fell victim to more traps, indicating a stronger vulnerability to keyword deception without prompt instructions.

The Power of Asymmetric Prompting over Parameter Count

The expanded 25-query benchmark fundamentally challenges the assumption that larger parameter counts directly correlate with better resilience against lexical deception. While the heavy `MPNet-Base` architecture successfully avoided 12 out of 25 traps, it incurred a significant computational penalty, averaging 31.24 ms per sentence.

Remarkably, the `e5-small-v2` model matched this accuracy (12/25 traps avoided) while operating at nearly a third of the latency (10.72 ms). Furthermore, it was the only model in the benchmark to maintain a positive average semantic margin (+0.0061). Standard bi-encoders compute the standard scaled dot-product attention **[14]** for a sequence of tokens independently:

$$\text{Attention}(Q_{mat}, K, V) = \text{softmax}\left(\frac{Q_{mat}K^T}{\sqrt{d_k}}\right)V \quad (3)$$

where the query, key, and value matrices (Q_{mat}, K, V) are derived entirely from the same input sentence. Because the evaluation query Q and the trap L are encoded in strict isolation, there is no cross-attention mechanism allowing the tokens of Q to interact with the negations in L. Consequently, the high token overlap dominates the pooling operation, resulting in vectors that are closely aligned in the hyperspace. This vulnerability of dense retrievers to exact lexical overlaps and "hard negatives" lacking cross-attention has been a well-documented limitation in neural ranking literature [15].

We attribute the success of e5-small-v2 to its prompt-instructed design. By requiring specific prefixes ("*query*:" and "*passage*:"}), the E5 architecture shifts from simple symmetric keyword matching to an asymmetric intent-resolution paradigm, a concept formalized by recent advancements in instruction-finetuned embeddings [16], making it highly robust against negated lexical traps.

Context Optimization Penalties

Notably, `nomic-embed-v1.5`, designed for massive 8192-token contexts, performed poorest on our short-sentence benchmarks. It exhibited the highest latency (44.08 ms) and failed 88% of the lexical traps (avoiding only 3 out of 25), with a severe negative margin of -0.1554. This highlights a critical architectural mismatch: long-context models introduce massive computational overhead and lose precision when applied to standard enterprise micro-queries.

Conclusion

Integrating semantic AI locally within enterprise C# applications via ONNX Runtime is highly viable, completely eliminating the privacy risks associated with cloud APIs. Our expanded benchmark across 25 lexical traps concludes that model selection must be strictly aligned with both hardware constraints and query structure, rather than relying solely on theoretical leaderboard rankings.

For edge devices requiring maximum throughput, the **quantized** MiniLM-L6 offers highly competitive latency ($\sim$ 3.3 ms). However, for standard .NET RAG architectures requiring true semantic comprehension and resilience against deceptive keywords, `e5-small-v2` represents the definitive architectural sweet

spot. Its prompt-instructed design ("*query*:"} vs "passage:") allows it to outperform significantly larger models, maintaining a positive semantic margin at a fraction of the computational cost (~ 10.7 ms).

Future work will explore the integration of hardware acceleration via DirectML for the ONNX execution provider. Furthermore, we plan to investigate how secondary cross-encoder reranking layers [17] can be implemented entirely in C# to further mitigate lexical deception. While cross-encoders introduce significantly higher latency by computing joint attention over the query and document simultaneously, pairing them with an ultra-fast quantized bi-encoder (like MiniLM-L6) could offer a hybrid .NET RAG pipeline that maximizes both computational performance and semantic accuracy.

Acknowledgments

This study is financed by the project No FP25-FMI-010 "Innovative Interdisciplinary Research in Informatics, Mathematics, and Pedagogy of Education" of the Scientific Fund of the Paisii Hilendarski University of Plovdiv, Bulgaria.

References

[1] P. Lewis et al., Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, *Advances in Neural Information Processing Systems (NeurIPS)*, 2020, Vol. 33, 9459–9474, ISSN: 1049-5258

[2] P. Hacker, A. Engel, M. Mauer, Regulating ChatGPT and other Large Generative AI Models, *Proc. of the 2023 ACM Conference on Fairness, Accountability, and Transparency (FAccT)*, 2023, 1112–1123, doi: 10.1145/3593013.3594067

[3] Microsoft, ONNX Runtime: Cross-platform ML inferencing, GitHub, 2024, [Online], available: https://github.com/microsoft/onnxruntime (last accessed: 08.05.2026)

[4] N. Muennighoff et al., MTEB: Massive Text Embedding Benchmark, *Proc. EACL*, 2023, 2014–2037, doi: 10.18653/v1/2023.eacl-main.148

[5] P. Kyurkchiev, Supplementary Dataset: Categorized Lexical Traps and Semantic Search Benchmark Logs, *Zenodo*, 2026, doi: 10.5281/zenodo.18953021

[6] N. Reimers, I. Gurevych, Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks, *Proc. EMNLP*, 2019, 3982–3992, doi: 10.18653/v1/D19-1410

[7] V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, *arXiv preprint arXiv: 1910. 01108*, 2019, doi: 10.48550/arXiv.1910.01108

[8] S. Xiao et al., C-Pack: Packaged Resources To Advance General Chinese Embedding, *arXiv preprint arXiv: 2309.07597*, 2023, (BGE Models), doi: 10.48550/arXiv.2309.07597

[9] Z. Nussbaum et al., Nomic Embed: Training a Reproducible Long Context Text Embedder, *arXiv preprint arXiv: 2402.01613*, 2024, doi: 10.48550/arXiv.2402.01613

[10] L. Wang et al., Text Embeddings by Weakly-Supervised Contrastive Pre-training, *arXiv preprint arXiv: 2212.03533*, 2022, doi: 10.48550/arXiv.2212.03533

[11] J. Chen, X. Ran, Deep Learning With Edge Computing: A Review, *Proc. of the IEEE*, 2019, Vol. 107, No. 8, 1655–1674, doi: 10.1109/ JPROC.2019.2921977, ISSN: 0018-9219

[12] Y. Kim et al., I-BERT: Integer-only BERT Quantization, *International Conference on Machine Learning (ICML)*, PMLR, 2021, Vol. 139, 5506–5518, ISSN: 2640-3498

[13] N. Thakur et al., BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models, *arXiv preprint arXiv: 2104.08663*, 2021, doi: 10.48550/arXiv.2104.08663

[14] A. Vaswani et al., Attention is all you need, *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, Vol. 30, 5998–6008, ISSN: 1049-5258

[15] J. Zhan, J. Mao, Y. Liu, J. Guo, M. Zhang, S. Ma, Optimizing Dense Retrieval Model Training with Hard Negatives, *Proc. SIGIR*, 2021, 1503–1512, doi: 10.1145/3404835.3462880

[16] H. Su et al., One Embedder, Any Task: Instruction-Finetuned Text Embeddings, *Findings of the Association for Computational Linguistics: ACL 2023*, 2023, 11022–11059, doi: 10.18653/v1/ 2023.findings-acl.71

[17] R. Nogueira, K. Cho, Passage Re-ranking with BERT, *arXiv preprint arXiv:1901.04085*, 2019, doi: 10.48550/arXiv.1901.04085

Pavel Kyurkchiev[1,*], Anton Iliev[1], Asen Rahnev[1], Viktor Matanski[1]

[1] Paisii Hilendarski University of Plovdiv,

Faculty of Mathematics and Informatics,

236 Bulgaria Blvd., 4027 Plovdiv, Bulgaria

Corresponding author: pkyurkchiev@uni-plovdiv.bg