Embrix: A Node.js Framework for Embedding Vector Generation and Similarity Evaluation (System Description and Initial Benchmarking Results)

Tahsin Özgür Koç

February 19, 2026

Abstract

I present Embrix, a lightweight Node.js framework I built for testing embedding inference: it generates sentence embedding vectors, measures latency/throughput, and checks semantic similarity using cosine distance. In a CPU-only Windows x64 environment (Node.js v25.5.0), I benchmark Embrix's current pipeline using two embedding models, MiniLM and BGE. For MiniLM, cold-start latency (first inference including model load) averages 30.18ms across five trials, with one outlier run at 140.10ms; warm-start inference over 50 iterations averages 2.68ms per embedding. Batching improves throughput and reaches 652 embeddings/s at batch size 100 in an end-to-end test; a dedicated batch-scaling sweep peaks at 696.9 embeddings/s at batch size 20.

To validate similarity behavior, I analyze cosine similarity for paraphrases and unrelated sentence pairs. MiniLM achieves mean similarity 0.902 for same-meaning pairs versus 0.0548 for unrelated pairs, yielding a separation margin of 0.217. In a small model comparison, both MiniLM and BGE obtain clustering purity 1.0 on a two-cluster toy dataset; MiniLM is faster (mean 2.56ms) than BGE (mean 4.28ms) and shows a slightly larger separation margin (0.204 vs. 0.182). Overall, this paper documents Embrix's current capabilities and provides baseline measurements to guide future development.

1 Introduction

Embedding models are a common building block for semantic search, clustering, retrieval-augmented generation, and similarity-based monitoring. In many systems, embedding inference runs on the critical path (e.g., request-time query embeddings) or on high-throughput pipelines (e.g., indexing corpora). Consequently, practitioners must reason about (i) cold-start latency, (ii) steady-state latency, (iii) batching behavior and throughput scaling, and (iv) whether the resulting representations are semantically stable enough to support downstream decisions.

This paper documents the current state of Embrix—a Node.js embedding vector creator and similarity checker that I built—and reports initial benchmark results. I focus on three questions:

1. **Cold vs. warm latency:** How large is the first-inference overhead relative to steady state?
2. **Batch scaling:** How does throughput change with batch size, and where do diminishing returns begin?
3. **Semantic stability:** Do embeddings meaningfully separate paraphrases from unrelated sentences, and are repeated encodings consistent?

I also include a limited **model comparison** between MiniLM and BGE to illustrate how latency and separation trade off across choices in Embrix.

2 System Architecture

Embrix consists of: (1) a *benchmark driver* that generates workloads (single-sentence and batched requests), (2) a *model runtime* that loads an embedding model and exposes an `embed` API, (3) a *measurement layer* that captures durations and memory usage, and (4) an *analysis layer* that summarizes runs into JSON artifacts.

2.1 Components

Benchmark driver. The benchmark driver is responsible for constructing the workload (single sentences or batches), running repeated trials, and orchestrating the experiment phases (cold start, warm start, and scaling sweeps). The driver also tags runs with metadata (timestamp, platform, and runtime versions) to make results comparable across iterations of Embrix.

Embedding runtime. The embedding runtime abstracts the underlying model implementation behind a minimal interface: load a model and compute an embedding vector for one sentence or a batch. This design allows Embrix to swap models while keeping the rest of the system (timing, reporting, and similarity analysis) unchanged.

Measurement and reporting. For each run, Embrix records wall-clock durations and memory snapshots before and after key phases (e.g., before model load, after model load, and after batched execution). It then aggregates these raw measurements into summary statistics (mean, median, variance, and percentiles) and stores them as JSON artifacts to support reproducibility and downstream analysis.

Similarity checker. Embrix includes a similarity checker that computes cosine similarity for selected sentence pairs. This provides a compact regression target: if a change in runtime, model, or preprocessing pipeline causes similarity distributions to shift significantly, Embrix can flag potential semantic drift.

Runtime environment. All experiments were run on Windows (x64) with Node.js v25.5.0. Embrix records timestamps and aggregates run-level statistics (mean, median, percentiles, and standard deviation).

Workloads. I measure (i) cold-start latency by timing the first inference in a fresh process/model-load context, (ii) warm-start latency by repeating inference after the model is loaded (50 iterations), (iii) throughput by running batched embedding generation over a range of batch sizes, and (iv) semantic stability via cosine similarity for paraphrases, unrelated pairs, and repeated identical sentences.

Outputs. Each experiment emits a JSON report containing raw measurements and derived statistics; this paper's sections are written directly from those artifacts.

3 Mathematical Formulation

3.1 Latency and throughput

Let n denote the batch size (number of sentences embedded in one call). We model end-to-end time as

$$T_{\text{total}}(n) = T_{\text{init}} + n\,T_{\text{embed}},$$

where T_{init} captures fixed overheads (initialization, dispatch, framework overhead) and T_{embed} is the marginal per-item cost. Fitting this model to measured batch latencies yields $T_{\text{init}} \approx 3.55\text{ms}$ and $T_{\text{embed}} \approx 1.50\text{ms}$ with $R^2 \approx 0.998$.

3.2 Stability and drift indicators

To track stability over time (e.g., across Embrix versions), I treat similarity as a random variable induced by the workload distribution. In addition to the separation margin Δ, two practical indicators are:

$$\mu_{\text{same}} = \mathbb{E}[\cos(u,v) \mid \text{same meaning}], \qquad \mu_{\text{unrel}} = \mathbb{E}[\cos(u,v) \mid \text{unrelated}].$$

We define throughput as

$$\text{Throughput}(n) = \frac{n}{T_{\text{total}}(n)} \quad \text{(embeddings/ms)}.$$

I report throughput in embeddings/s.

3.3 Cosine similarity and semantic separation

Given embeddings $u, v \in \mathbb{R}^d$, we compute cosine similarity

$$\cos(u, v) = \frac{u^\top v}{\|u\|_2 \, \|v\|_2}.$$

To summarize discrimination between related and unrelated pairs, we use a separation margin

$$\Delta = \mathbb{E}[\cos(u, v) \mid \text{same meaning}] \; - \; \mathbb{E}[\cos(u, v) \mid \text{unrelated}].$$

4 Experimental Setup

Harness and models. Experiments are executed in Embrix. I primarily evaluate MiniLM, and include a comparison against BGE for latency and semantic separation.

Models. **MiniLM** serves as the default baseline model due to its low latency. **BGE** is included as a second point in the design space to test how Embrix behaves when swapping models and to compare latency and separation margins under the same harness.

Hardware and runtime. The experiments are executed in a CPU-only Windows x64 environment with Node.js v25.5.0. Embrix records process memory metrics (heap and RSS) around load and inference phases to quantify overheads that matter in long-running services.

Latency measurement. Cold-start latency is measured over 5 runs as the time to first inference including model load. Warm-start latency is measured over 50 iterations after the model is loaded.

Batch scaling. We evaluate throughput across batch sizes from 1 to 200 (three iterations per batch size in the batch-scaling experiment), recording both time and memory deltas.

Semantic stability. We construct (i) paraphrase test cases (same meaning), (ii) unrelated sentence pairs, and (iii) repeated identical-sentence encodes to verify consistency. Similarities are computed using cosine similarity in embedding space.

Reproducibility notes. Because the goal of this paper is to describe Embrix's current state, the reported numbers should be interpreted as a baseline rather than a universal benchmark. Small changes in runtime version, background system load, or model caching can affect cold-start behavior. For that reason, Embrix stores raw samples in addition to aggregate summaries so that outliers (e.g., an unusually high cold-start run) can be inspected rather than silently averaged away.

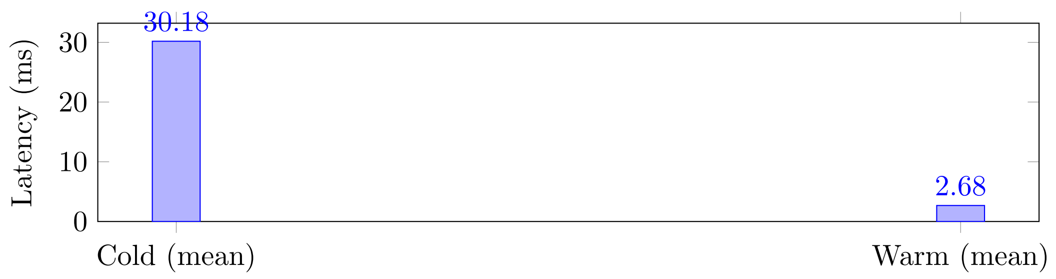

Figure 1: MiniLM cold-start vs. warm-start mean latency in Embrix.

5 Results

5.1 Cold vs. Warm Latency

For MiniLM, cold-start latency (first inference including model load) has a mean of 30.18ms across five trials, but with substantial variance driven by a single outlier run at 140.10ms. Excluding this outlier, the remaining cold-start measurements fall between 2.19ms and 3.12ms. Warm-start latency over 50 iterations averages 2.68ms per embedding (median 2.75ms), indicating stable steady-state performance.

Interpreting the outlier. The cold-start outlier suggests that, in practice, initialization overhead is not always constant. Potential causes include filesystem caching effects, JIT warm-up, and transient CPU scheduling delays. For Embrix, this motivates reporting not only the mean but also robust statistics (median and percentiles) when making engineering decisions.

5.2 Batch Scaling

Throughput improves with batching and reaches its maximum in the batch-scaling sweep at batch size 20 with 696.9 embeddings/s (average 1.43ms per item). Beyond this point, throughput plateaus and fluctuates: for example, batch sizes 75–200 remain in the $\sim$ 535–556 embeddings/s range. A quadratic batch-time model

$$T_{\text{batch}}(n) = \alpha n^2 + \beta n + \gamma$$

fits the measured scaling curve with $R^2 \approx 0.997$.

Operational guidance. In Embrix's current results, batch sizes in the 10–50 range provide consistently strong throughput, while very large batches do not reliably improve performance. This pattern supports a simple batching policy for services: accumulate small micro-batches to improve utilization, but cap batch sizes to limit queueing delay and variance.

Pair type	Mean cosine similarity	Notes
Same meaning (paraphrase)	0.902	higher is better
Unrelated	0.0548	lower is better
Separation margin Δ	0.217	difference of means

Table 1: Cosine similarity summary for MiniLM in Embrix (toy sentence sets).

Model	Mean latency (ms)	Purity	Separation margin
MiniLM	2.56	1.00	0.204
BGE	4.28	1.00	0.182

Table 2: MiniLM vs. BGE comparison summary in Embrix (toy evaluation).

5.3 Semantic Stability

Across paraphrase test cases, MiniLM achieves an aggregate average cosine similarity of 0.902 for same-meaning pairs. For unrelated sentence pairs, the average similarity is 0.0548. The resulting separation margin is $\Delta \approx 0.217$, indicating clear discrimination in this controlled evaluation. For repeated encodings of an identical sentence, pairwise cosine similarity is 1.0 throughout (perfect stability in the harness).

Threshold selection. Embrix's similarity checker is often used with simple thresholds (e.g., "treat two sentences as similar if cosine similarity exceeds τ"). The observed gap between paraphrase and unrelated pairs suggests that, for this toy workload, a threshold in the range $\tau \in [0.5, 0.8]$ would separate the two sets comfortably. In practice, Embrix is intended to help calibrate τ on a task-specific dataset rather than relying on a single global value.

5.4 Model Comparison

In a two-cluster toy evaluation, both MiniLM and BGE achieve clustering purity of 1.0. MiniLM is faster in steady state (mean 2.56ms per embedding) than BGE (mean 4.28ms). MiniLM also shows a slightly larger separation margin (0.204) than BGE (0.182) in the comparison report, suggesting marginally stronger semantic separation on this dataset, while BGE is selected as "winner" in the summary artifact.

Interpreting "winner". Embrix's summary report marks BGE as the winner, but the choice depends on the objective function: if the primary constraint is latency, MiniLM is preferable in this environment; if the downstream objective weights other quality dimensions not captured by the toy clustering task, a slower model may be justified. A key goal of Embrix is to make these tradeoffs explicit and measurable.

6 Discussion

The cold-start results indicate that one-time initialization effects can dominate tail latency when models are loaded on demand; in Embrix, this motivates process reuse, explicit model preloading, or a warm worker pool. For steady-state operation, MiniLM provides sub-3ms mean latency per embedding in my environment and benefits substantially from batching up to moderate batch sizes. The batch-scaling sweep suggests a practical operating point near batch size 20 for peak throughput, while larger batches provide limited additional throughput and can complicate scheduling.

The semantic stability metrics show a clear gap between paraphrase similarity and unrelated similarity, supporting the use of Embrix's similarity checks for small-scale testing and regression monitoring. That said, the results are based on toy sentence sets and a limited two-model comparison; broader datasets and downstream task evaluations are required before drawing stronger conclusions about model quality.

6.1 Limitations

The experiments in this paper are intentionally lightweight and designed to validate Embrix end-to-end rather than to establish definitive state-of-the-art numbers. The sentence sets are small, and the clustering evaluation uses a toy two-topic dataset. In addition, CPU-only results may not reflect performance in GPU-backed production systems.

6.2 Practical use cases

The current Embrix workflow is particularly useful in three scenarios: (i) quickly comparing embedding models under a fixed runtime and deployment environment, (ii) regression testing after upgrading Node.js or changing preprocessing code, and (iii) calibrating similarity thresholds for a specific application by observing the distribution of cosine similarities for positive and negative pairs.

7 Conclusion

I documented the current state of Embrix and provided an initial experimental characterization of embedding inference for MiniLM, with a small comparison against BGE. The results highlight mean warm-start latency of 2.68ms for MiniLM, peak batch-scaling throughput of 696.9 embeddings/s at batch size 20, and strong separation between paraphrase and unrelated pairs ($\Delta \approx 0.217$). These measurements serve as a baseline for future Embrix releases and for selecting embedding models under latency constraints.