Embedding Models for RAG: Which to Run Locally

February 8, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** For most local RAG setups, nomic-embed-text (137M parameters, 274MB download, 8K context) is still the right default — it runs on CPU, fits alongside any chat model, and produces good retrieval quality. The new option worth knowing: Qwen3-Embedding 0.6B scores dramatically higher on benchmarks (70.7 MTEB v2) with 32K context and 1.2GB download, also in Ollama. For multilingual documents, bge-m3 supports 100+ languages. For absolute minimum hardware, all-minilm is 46MB and fast enough for small collections. All of these run comfortably on CPU — embedding models don't need a GPU.

📚 **Related:** Local RAG Guide · Best LLMs for RAG · AnythingLLM Setup · Open WebUI Setup · Planning Tool

Your RAG pipeline has two models: the chat model that generates answers, and the embedding model that finds the right chunks to feed it. Most people spend all their time picking the chat model and accept whatever embedding default their tool ships with.

That's backwards. A bad embedding model means the wrong chunks get retrieved, and even a 70B chat model can't answer correctly from irrelevant context. The embedding model is the retrieval engine — if it fails, everything downstream fails.

The good news: embedding models are tiny. The largest one you'd realistically run locally is 1.2GB. Most are under 700MB. They run on CPU without noticeable slowdown. And they're free to swap — re-embed your documents and you're done.

Here's what to run, why, and how to avoid the mistakes that break retrieval quality.

What Embedding Models Do

Embedding models convert text into vectors — arrays of numbers (typically 384 to 4,096 numbers) that capture semantic meaning. Similar meanings produce similar vectors.

"How do I fix a memory leak?" and "troubleshooting RAM issues in my application" produce vectors that are close together in vector space — even though they share no words. "The weather is nice today" produces a vector far away from both.

This is what powers semantic search. Instead of matching keywords, your RAG system finds chunks whose meaning is closest to your question.

The process:

1. Split your documents into chunks
2. Run each chunk through the embedding model → get a vector per chunk
3. Store vectors in a vector database (ChromaDB, FAISS, etc.)
4. When you ask a question, embed the question → find the closest vectors → retrieve those chunks
5. Feed the chunks to your chat model as context

The embedding model runs twice: once during indexing (every chunk) and once per query (your question). Indexing is a one-time cost. Queries are fast — typically under 50ms for a single question.

The Models Worth Running

nomic-embed-text — The Default Choice

nomic-embed-text v1.5 has become the de facto standard for local RAG. It hits the sweet spot: small enough to run alongside any chat model, large enough context to handle real documents, and good enough quality that most people never need to upgrade.

Spec	Value
Parameters	137M
Dimensions	768
Max tokens	8,192
MTEB English	62.3
License	Apache 2.0
Ollama download	274 MB

Why it's the default: 8K token context means you can embed large chunks without truncation. 274MB download means it loads in seconds and sits comfortably in memory alongside a 7B or 14B chat model. Matryoshka support lets you shrink dimensions (512, 256, 128, 64) to save storage if you're embedding millions of chunks.

One quirk: nomic-embed-text uses task prefixes. When embedding documents, prepend `search_document:` to the text. When embedding queries, prepend `search_query:`. Ollama and most RAG tools handle this automatically, but if you're using the model directly via API, missing the prefix hurts retrieval quality.

```
ollama pull nomic-embed-text
```

Qwen3-Embedding 0.6B — The New Benchmark

Released June 2025, Qwen3-Embedding rewrote the benchmark leaderboards. The 0.6B model scores 70.7 on MTEB English v2 — dramatically higher than models twice its size from the previous generation. And it's in Ollama.

Spec	Value
Parameters	0.6B
Dimensions	1,024 (Matryoshka: 32-1,024)
Max tokens	32,000
MTEB English v2	70.7
License	Apache 2.0
Ollama download	1.2 GB

Why it matters: 32K token context is huge — you can embed entire documents as single chunks if your use case benefits from that. The quality gap over nomic-embed-text is significant on benchmarks, especially for complex queries. Multilingual support covers 100+ languages.

The catch: At 1.2GB, it's 4x the size of nomic-embed-text. Still tiny by LLM standards, but on an 8GB VRAM card running a 7B chat model, that extra gigabyte matters. Also newer — less community testing and fewer integrations handle its task instruction format automatically.

Larger variants exist (4B at 8GB, 8B at 15GB) but they're impractical for most local setups — they compete with your chat model for VRAM.

```
ollama pull qwen3-embedding:0.6b
```

mxbai-embed-large — Highest Quality (English)

mxbai-embed-large consistently scores at the top of English retrieval benchmarks among models under 500M parameters. If your documents are English-only and retrieval precision matters more than speed, this is the upgrade from nomic.

Spec	Value
Parameters	335M
Dimensions	1,024
Max tokens	512
MTEB English	64.7
License	Apache 2.0
Ollama download	670 MB

The limitation: 512 token max context. Your chunks must stay under ~400 words, or the embedding only captures the first half and silently ignores the rest. This is a real constraint — it forces smaller chunks, which can fragment context that spans multiple paragraphs.

If your documents have short, self-contained sections (FAQs, API docs, product descriptions), this works great. If your documents have long, flowing arguments (research papers, legal text), the 512-token limit hurts.

```
ollama pull mxbai-embed-large
```

bge-m3 — Best for Multilingual

bge-m3 from BAAI supports 100+ languages and 8K token context. The “M3” stands for Multi-linguality, Multi-granularity, and Multi-functionality — it produces dense, sparse, and ColBERT-style multi-vector embeddings simultaneously.

Spec	Value
Parameters	568M
Dimensions	1,024
Max tokens	8,192
License	MIT
Ollama download	1.2 GB

When to use it: Your documents include non-English text. Period. For English-only RAG, nomic-embed-text or mxbai-embed-large produce better retrieval. bge-m3 trades English accuracy for breadth across languages.

Hybrid retrieval: bge-m3 generates both dense vectors (semantic) and sparse vectors (keyword-like) in a single pass. If your RAG framework supports hybrid search (Qdrant and Milvus do), you can use both signals for better retrieval than either alone.

```
ollama pull bge-m3
```

all-minilm — The Tiny Option

all-MiniLM-L6-v2 is the smallest usable embedding model. At 22.7M parameters and 46MB, it runs on anything — Raspberry Pi, old laptops, CI/CD pipelines.

Spec	Value
Parameters	22.7M
Dimensions	384
Max tokens	256
MTEB English	~56
License	Apache 2.0
Ollama download	46 MB

The trade-offs: 256-token context is extremely short — about 200 words per chunk. Retrieval quality is noticeably worse than larger models. You'll get more irrelevant chunks and miss more relevant ones.

When it's fine: Small personal document collections (< 1,000 chunks), resource-constrained devices, or situations where embedding speed matters more than precision. It embeds thousands of documents per second on CPU.

```
ollama pull all-minilm
```

EmbeddingGemma — Google's Entry

EmbeddingGemma (308M parameters, 622MB) is Google's purpose-built embedding model from the Gemma 3 architecture. Multilingual support for 100+ languages, QAT-optimized to run in under 200MB RAM on edge devices.

```
ollama pull embeddinggemma
```

A solid middle-ground option, especially if you're already in the Gemma ecosystem. The 2,048-token context is shorter than nomic-embed-text's 8K, which limits chunk size.

Comparison Table

Model	Params	Dims	Max Tokens	Quality	Ollama Size	Best For
all-minilm	22.7M	384	256	Acceptable	46 MB	Minimal hardware
nomic-embed-text	137M	768	8,192	Good	274 MB	**Default choice**
EmbeddingGemma	308M	768	2,048	Good	622 MB	Multilingual, Gemma stack
mxbai-embed-large	335M	1,024	512	Very good	670 MB	English precision
bge-m3	568M	1,024	8,192	Good (multi)	1.2 GB	Multilingual documents
Qwen3-Embedding 0.6B	0.6B	1,024	32,000	Excellent	1.2 GB	Maximum quality

"Quality" is a rough ranking based on MTEB scores, but note that Qwen3's scores are on MTEB v2 while older models were tested on v1 — not directly comparable. In practice, nomic-embed-text produces good results for most RAG use cases.

VRAM and RAM Requirements

Embedding models are small enough that VRAM is rarely the bottleneck. Here's what they actually need:

Model	Memory (loaded)	Runs on CPU?	Speed Impact
all-minilm	~90 MB	Yes, fast	Negligible
nomic-embed-text	~300 MB	Yes, fast	Negligible
EmbeddingGemma	~600 MB	Yes	Minor
mxbai-embed-large	~700 MB	Yes	Minor
bge-m3	~1.2 GB	Yes	Noticeable on slow CPUs
Qwen3-Embedding 0.6B	~1.2 GB	Yes, slower	Noticeable on slow CPUs

Key point: You don't need GPU for embeddings. CPU inference is fast because embedding models are tiny. The only time GPU matters is bulk indexing — embedding 100,000 documents goes faster on GPU. For individual queries (one embedding per question), CPU handles it in milliseconds.

Running alongside your chat model: Ollama loads the embedding model separately from your chat model. On an 8GB card, a 7B chat model (~5GB) plus nomic-embed-text (~300MB) fits comfortably. With Qwen3-Embedding 0.6B (~1.2GB), you're tighter but still fine. On 12GB+, any combination works.

How to Use Embeddings With Ollama

Generate an Embedding

```
curl http://localhost:11434/api/embed -d '{
  "model": "nomic-embed-text",
```

```
  "input": "Your text to embed"
}'
```

The response includes a `"embeddings"` array — one vector per input string.

Batch Embedding

```
curl http://localhost:11434/api/embed -d '{
  "model": "nomic-embed-text",
  "input": [
    "First chunk of text",
    "Second chunk of text",
    "Third chunk of text"
  ]
}'
```

Batching is faster than individual calls. When embedding documents, send chunks in batches of 50-100.

Python Example

```
import ollama

# Embed a single text
response = ollama.embed(
    model="nomic-embed-text",
    input="What are the quarterly revenue figures?"
)
vector = response["embeddings"][0]
print(f"Dimensions: {len(vector)}")  # 768 for nomic-embed-text

# Embed multiple texts
response = ollama.embed(
    model="nomic-embed-text",
    input=["chunk one", "chunk two", "chunk three"]
)
vectors = response["embeddings"]  # List of 3 vectors
```

In RAG Tools

Most RAG tools handle embeddings automatically:

- **Open WebUI:** Settings → Documents → Embedding Model → select your Ollama model
- **AnythingLLM:** Settings → Embedding → Provider: Ollama → Model: nomic-embed-text
- **LangChain:** `OllamaEmbeddings(model="nomic-embed-text")`

You configure the model name once. The tool handles calling the API, batching, and storing vectors.

Chunking: Where Most RAG Breaks

The embedding model converts chunks to vectors. If your chunks are bad, your vectors are bad, and retrieval fails no matter how good the model is.

Chunk Size Guidelines

Embedding Model	Max Tokens	Recommended Chunk Size
all-minilm	256	150-200 tokens
mxbai-embed-large	512	300-450 tokens
nomic-embed-text	8,192	500-1,000 tokens
bge-m3	8,192	500-1,000 tokens
Qwen3-Embedding 0.6B	32,000	500-2,000 tokens

Critical rule: Your chunks must be shorter than your embedding model's max token limit. If a 1,000-token chunk hits a model with a 512-token limit, the model only embeds the first 512 tokens. The second half is invisible to search — and you won't get an error message.

Too Large vs Too Small

Chunks too large (2,000+ tokens):

- The embedding averages over too many topics
- Retrieval returns chunks that partially match but contain mostly irrelevant text
- The chat model gets noise alongside signal

Chunks too small (100-200 tokens):

- Individual sentences lack context
- "Revenue increased 15%" means nothing without knowing which quarter, which product, which region
- The chat model gets fragments that can't support a useful answer

Sweet spot: 500-1,000 tokens with 10-20% overlap between consecutive chunks. This preserves context within each chunk while keeping the embedding focused enough for precise retrieval.

Overlap Matters

When you split a document at chunk boundaries, information that spans two chunks gets cut in half. A 10-20% overlap means the end of chunk N appears again at the start of chunk N+1. This prevents sentences from being orphaned.

```
Chunk 1: [tokens 1-1000]
Chunk 2: [tokens 900-1900]    ← 100 tokens overlap
Chunk 3: [tokens 1800-2800]   ← 100 tokens overlap
```

Most RAG tools let you configure overlap. Set it to 10-20% of chunk size.

Vector Databases

You need somewhere to store the vectors. These are the options that work locally.

Database	Setup	Best For	Storage
ChromaDB	`pip install chromadb`	Simple, getting started	Disk (SQLite)
LanceDB	Built into AnythingLLM	Zero-config	Disk (Lance format)
FAISS	`pip install faiss-cpu`	Large collections, speed	In-memory
Qdrant	Docker or binary	Production, hybrid search	Disk + memory
pgvector	PostgreSQL extension	Existing Postgres users	PostgreSQL

For most people: ChromaDB or LanceDB. They're built into the popular RAG tools (Open WebUI uses ChromaDB, AnythingLLM uses LanceDB), require zero configuration, and handle document collections in the thousands without breaking a sweat.

When to upgrade: If you have 100K+ chunks, need hybrid search (dense + sparse vectors), or want production features like snapshots and replication, move to Qdrant. It runs as a single Docker container and has a Python client.

FAISS: Facebook's vector library. Extremely fast for similarity search — handles millions of vectors. The trade-off: in-memory by default (high RAM usage for large collections) and no built-in persistence without extra code. Best for large-scale batch processing.

Which Model for Which Situation

Situation	Model	Why
Just getting started with RAG	nomic-embed-text	Small, fast, good enough, well-supported
English docs, maximum retrieval quality	Qwen3-Embedding 0.6B	Highest benchmark scores, 32K context
Multilingual documents	bge-m3	100+ languages, 8K context
Extremely limited hardware	all-minilm	46MB, runs on anything
Short-form content (FAQs, product pages)	mxbai-embed-large	Best precision at 512-token chunks
Already using Gemma models	EmbeddingGemma	Consistent ecosystem, 100+ languages
Need hybrid search (dense + sparse)	bge-m3	Only model supporting both in one pass

The Practical Decision

If you're new to RAG: **use nomic-embed-text**. It's the most tested, most documented, and most compatible option. Every RAG tutorial uses it. Every tool supports it. It works.

If you're comfortable with RAG and want better retrieval: **try Qwen3-Embedding 0.6B**. The quality improvement is real, the 32K context removes chunk-size constraints, and it's in Ollama. The 1.2GB download is still small. Re-embed your documents and compare results.

Common Mistakes

Using Your Chat Model for Embeddings

Chat models (Qwen 2.5 7B, Llama 3.1 8B) can technically produce embeddings, but they're worse at retrieval than purpose-built embedding models. A 137M embedding model outperforms a 7B chat model for vector search — it was trained specifically for similarity, the chat model wasn't. And it's 30x smaller.

Mixing Embedding Models

Vectors from different models are incompatible. If you embed half your documents with nomic-embed-text and half with mxbai-embed-large, similarity search between the two halves is meaningless — the vector spaces don't align. Pick one model and use it for everything. If you switch models, re-embed all documents.

Ignoring the Token Limit

This is the silent killer. If your chunks are 1,500 tokens and your embedding model caps at 512, every chunk gets truncated to 512 tokens before embedding. The model can't see the rest. Retrieval degrades and you don't get an error — just worse results.

Check your embedding model's max token limit. Keep chunks below it.

Chunks Too Large

A 3,000-token chunk about "Q3 revenue, hiring plans, and product roadmap" produces one vector that represents all three topics. A query about Q3 revenue matches that chunk — but so does a query about hiring, and the chat model gets 2,000 tokens of irrelevant context. Smaller, focused chunks produce more precise retrieval.

Skipping Overlap

Without overlap, a sentence split across two chunks gets halved in both. "Revenue grew 15% in Q3, driven primarily by the new enterprise tier" might become "Revenue grew 15% in Q3, driven primarily" in one chunk and "by the new enterprise tier, which launched in July" in the next. Neither chunk contains the complete thought. Add 10-20% overlap.

Not Re-Embedding After Changes

If you add documents to an existing collection, they need to be embedded with the same model and settings. If you change chunk size, overlap, or embedding model, you need to re-embed everything — old and new documents — for consistent retrieval quality.

Speed: How Fast Are They?

Embedding speed varies by hardware, but here are rough numbers on a modern CPU (AMD Ryzen 7 / Intel i7):

Model	Single Query	Bulk Indexing (CPU)	Bulk Indexing (GPU)
all-minilm	< 5 ms	~500 chunks/sec	~2,000 chunks/sec
nomic-embed-text	< 10 ms	~100 chunks/sec	~500 chunks/sec
mxbai-embed-large	< 15 ms	~50 chunks/sec	~300 chunks/sec
bge-m3	< 20 ms	~30 chunks/sec	~200 chunks/sec
Qwen3-Embedding 0.6B	< 25 ms	~20 chunks/sec	~150 chunks/sec

For queries: All of these are fast enough. Under 25ms per query is imperceptible — the chat model's response takes 100x longer.

For initial indexing: A 10,000-chunk document collection takes ~100 seconds with nomic-embed-text on CPU, ~500 seconds with Qwen3-Embedding 0.6B. That's a one-time cost. After initial indexing, only new documents need embedding.

GPU acceleration helps for bulk indexing but isn't needed for query-time embedding. If you're indexing a huge document collection, load the embedding model on GPU for the initial pass, then run it on CPU for queries.

The Workflow

Here's how embedding models fit into a complete local RAG setup:

1. Choose Your Models

```
# Chat model (for generating answers)
ollama pull qwen2.5:14b

# Embedding model (for search)
ollama pull nomic-embed-text
```

2. Chunk Your Documents

Split documents into 500-1,000 token chunks with 10-20% overlap. Your RAG tool (Open WebUI, AnythingLLM) handles this automatically. If you're using Python, `RecursiveCharacterTextSplitter` from LangChain is the standard approach.

3. Embed and Store

Feed chunks through the embedding model, store vectors in your database. Again, tools handle this — upload a PDF, they chunk it, embed it, and store it.

4. Query

Type a question. The system embeds your question with the same model, finds the closest chunk vectors, retrieves those chunks, and passes them to the chat model as context.

5. Evaluate

If answers are wrong or incomplete, check:

- Are the right chunks being retrieved? (Most tools show retrieved sources)
- Are chunks too large or too small?
- Is the embedding model's token limit being exceeded?
- Would a better embedding model help?

For the full pipeline setup, see the local RAG guide or the AnythingLLM setup guide.

The Bottom Line

Start with nomic-embed-text. It's 274MB, runs on CPU, has 8K context, and is supported everywhere. For most personal and small-team RAG setups, it's all you need.

If retrieval quality matters enough to justify a larger model, Qwen3-Embedding 0.6B is the current leader — dramatically better benchmarks, 32K context, and still only 1.2GB. It's the upgrade path when nomic-embed-text isn't finding the right chunks.

The embedding model gets less attention than the chat model, but it determines whether the right information reaches the chat model in the first place. A 70B chat model with bad retrieval produces worse answers than a 7B chat model with good retrieval. Invest in the search side of RAG first.

📚 **RAG guides:** Local RAG Setup · Best LLMs for RAG · AnythingLLM Setup · Open WebUI Setup

📚 **Model guides:** VRAM Requirements · Qwen Models Guide · Ollama Setup

---