Embedding Model Inference

baseten

Table of Contents

PART 01

Introduction

Embedding models are the connective tissue of modern AI systems, powering semantic search, retrieval-augmented generation (RAG), recommender systems, and compound AI agents. From indexing millions of documents to handling a single high-value query in real time, embedding inference performance directly shapes product quality.

When embeddings are slow, search results feel stale. When throughput is low, ingestion pipelines stall. The challenge is clear: deliver both high throughput and low latency in production, without overspending on infrastructure.

This guide shows how to meet those demands, and how Baseten provides the fastest embedding model runtime in production today.

What are embeddings?

An embedding model transforms a variable-length chunk of text – or another modality of input like an image – into a fixed-length vector representation that captures the semantic meaning of the input. By encoding content into this shared semantic vector space, you can compare similarity between items with simple math.

The output vectors from embedding models vary in dimensionality, or the count of numbers in the vector, ranging from a few hundred to a few thousand dimensions. Higher-dimensionality vectors store more data, while lower-dimensionality vectors are faster and cheaper to store and process. Most modern embedding models use Matryoshka Representation Learning to enable a single model to encode information at various dimensionalities while retaining as much data as possible in smaller dimensions.

Embedding models are used for more than just RAG. These models and their outputs quietly power everything from upgrading legacy ML systems to building cutting-edge agents.

Embeddings are useful for:

- **Context:** The “R” in “RAG” is retrieval, and embeddings let you retrieve meaningful context for LLM prompts and agentic actions.
- **Memory:** Interface efficiently with memory by shifting information from the context window to a vector store.
- **Personalization:** Build user profiles by embedding data and activity.
- **Search:** Quickly scan massive corpuses for relevant results.
- **Classification:** Categorize items based on semantic similarity, or detect anomalies.
- **Recommender systems:** Recommend similar content or products.

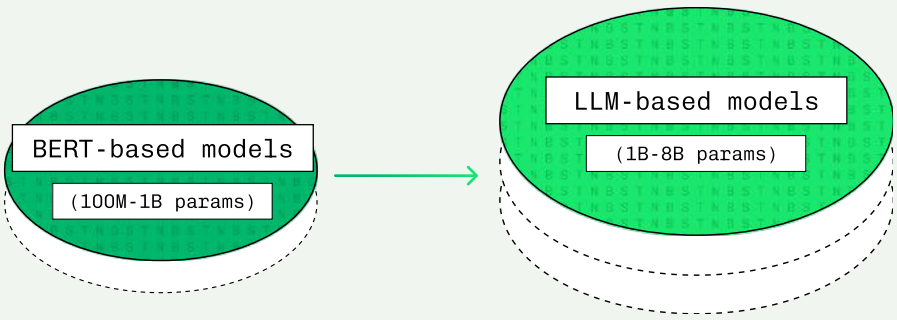
CONTINUED >

What are embeddings?

Building these systems takes more than just an embedding model. Once you have generated the embeddings, you need infrastructure to store and query them efficiently. That's where vector databases come in as specialized systems that support fast nearest-neighbor search across billions of vectors. There are a number of excellent vendor options on the market, from purpose-built vector databases like Chroma, Pinecone, Qdrant, and Weaviate, to vector support within larger ecosystems like AWS, MongoDB, PostgreSQL, and many others.

Just like you need a specialized database for storing and using embeddings, you also need specialized infrastructure for running the models. Previously, ML engineers trained regression and tree-based models for classification and recommendation, and AI engineers relied on techniques like fuzzy string matching and knowledge graph traversal for search.

But with the rise of models like BERT in 2018, language model-based embedding systems took over many of these workloads thanks to their increased accuracy and flexibility. Today, embedding models are often built from the same neural networks that power large language models like Mistral and Qwen, ranging from one to eight billion parameters. But unlike LLMs, embedding models are generally deterministic, taken directly from hidden states rather than sampling for inference.



These larger embedding models require more powerful hardware, especially for high-volume production inference. Where before a tree-based classifier may have run on a modestly-specced CPU, today's embedding models require GPUs like H100 or B200 for fast inference.

State-of-the-art open-source embedding models

Embedding models are among the smallest generative AI models by parameter count. This makes them relatively cheap to train, leading to a proliferation of open-source models available on the market.

AI engineers choose open-source models for a wide range of reasons. It often comes down to domain-specific quality, consistently low latency, strong unit economics, and zero platform risk. That last point is especially important for embedding models, where the output of one model is not compatible with other models. So if you build on a closed-source model but lose access due to a deprecation, you'll have to go through the time and expense of re-generating all of your previous embeddings in a new model.

On quality, open-source embedding models are up to par with closed-source options from vendors like OpenAI and Google Gemini. On top of strong out-of-the-box performance, embedding models' small size and straightforward architecture make them a great candidate for fine-tuning to improve domain-specific quality. Some teams even train their own frontier embedding models to power search or retrieval within their domain.

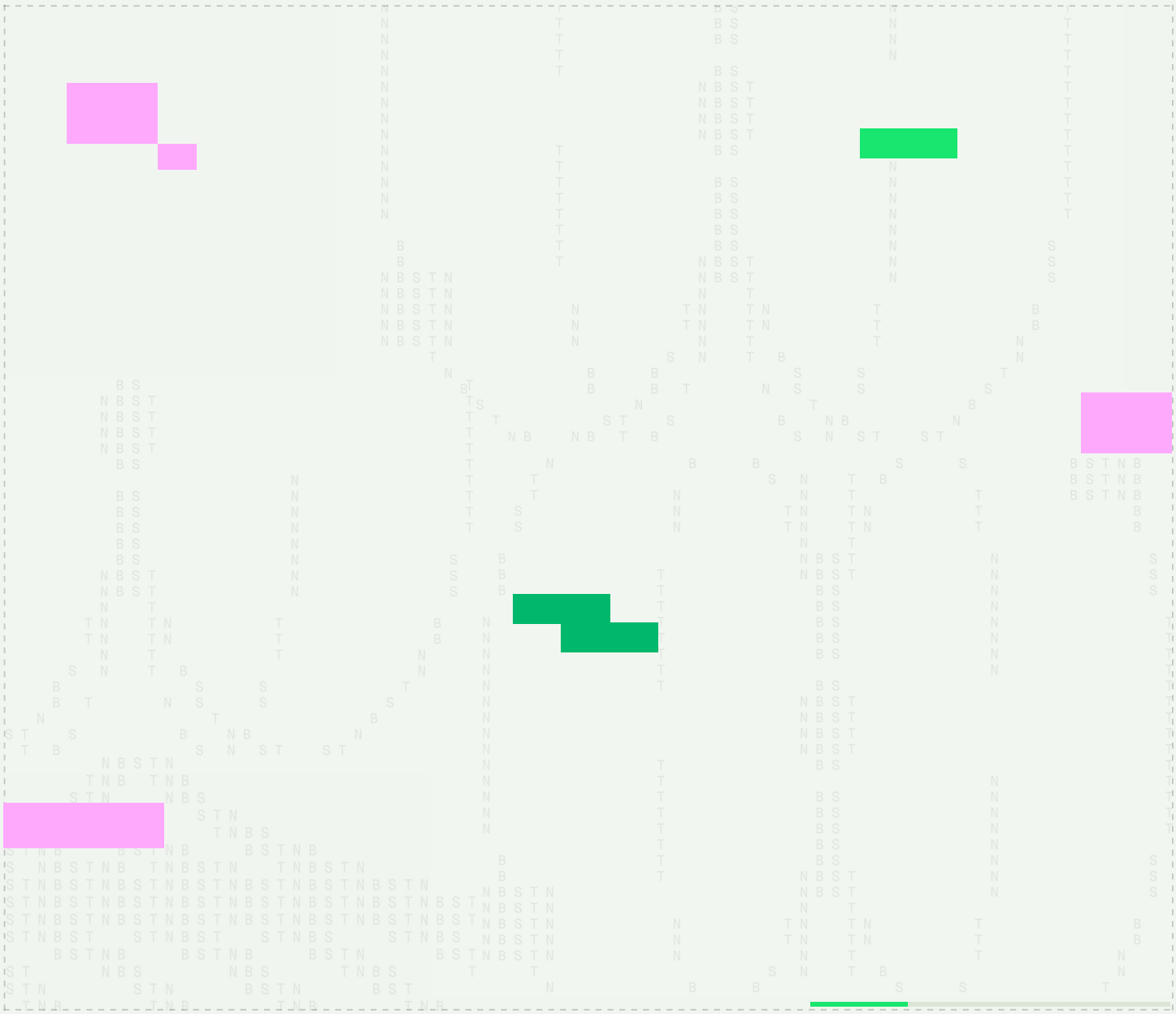
Strong starting points in the world of open-source embedding models include:

- **Qwen:** The open-source AI research lab within Alibaba consistently releases excellent general-purpose embedding models in a range of sizes.
- **Gemma:** Embedding models from Google's open-source Gemma lab offer frontier performance at small sizes.
- **BGE:** The BGE family of embedding models by BAAI (Beijing Academy of Artificial Intelligence) includes reranking in addition to embedding.
- Open-source options from startups like **Nomic**, **Jina**, **MixedBread**, and **ZeroEntropy**, which release multi-modal models and models for specific domains like coding.

CONTINUED >

State-of-the-art open-source embedding models

While closed-source embedding models from labs like OpenAI are a great way to get started and are cost-effective for low-traffic prototypes and side projects, production applications are better served by open-source models. However, to unlock the benefits of consistent low latency and cost-effective large-scale inference, you need an optimized model serving solution for your open-source or fine-tuned embedding model.



Why embedding inference is uniquely challenging

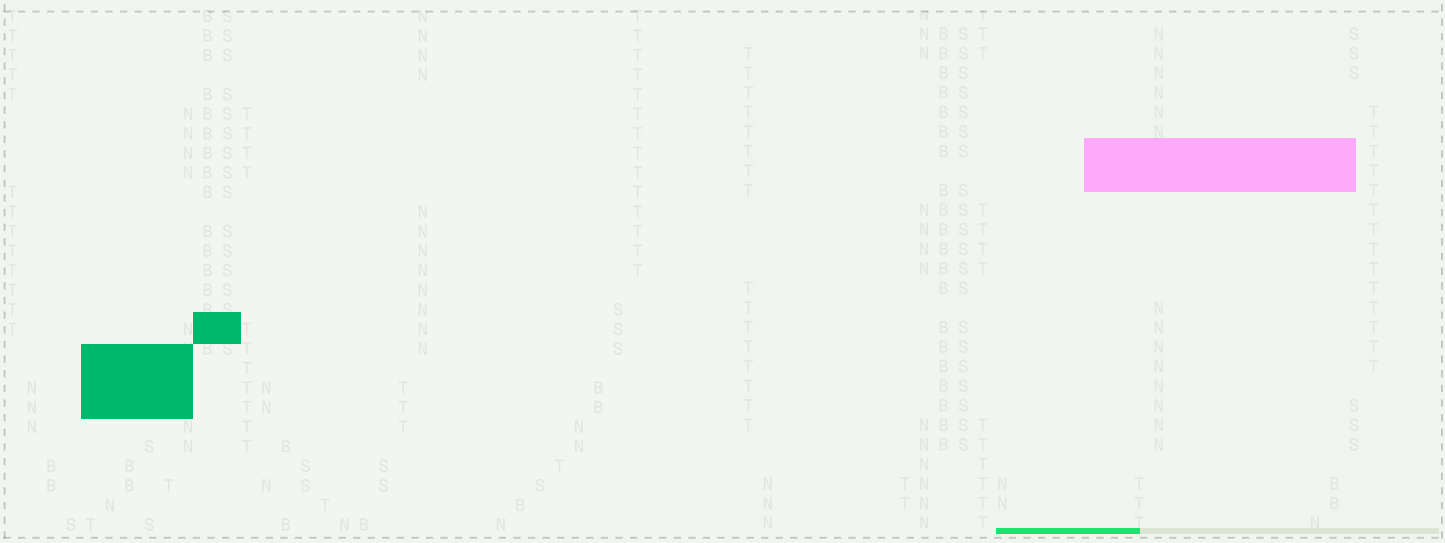
Embedding inference workloads have unique characteristics compared to serving other models like LLMs. Firstly, embedding inference systems need to support two very different traffic profiles:

1. **High-throughput backfills:** Bulk operations like indexing millions of documents, updating product catalogs, or even preparing data for LLM pre-training.
2. **Low-latency lookups:** Individual user-facing queries for search, retrieval, or recommendation, where every millisecond affects user experience.

Generally, inference performance optimization is discussed in terms of trading off latency for throughput or vice versa. But embedding jobs combine:

- High concurrency (thousands of simultaneous requests)
- Small input size but high frequency (short text, but lots of it)
- Small model size (1-8 billion parameters, but very large memory requirements for batching)
- Tight SLAs (especially for search and recommender systems)

While you can build systems that serve this dual profile, let's first consider how to optimize for each individually.



Optimizing for offline throughput

If you're configuring a model deployment only for throughput — for example, to support a database backfill by embedding an entire corpus — you mostly care about stability and cost. These "offline" or non-latency-sensitive jobs involve processing billions, or even trillions, of tokens of input data, and their runtime is generally measured in GPU hours rather than wall-clock time.

Some examples of offline embedding model workloads include:

- Large-scale search index creation
- Document corpus embedding for future retrieval
- Synthetic data processing for LLM pre-training
- Classification on massive datasets

Throughput optimization is the engineering work required to process this many requests without losing data, as cost-effectively as possible. This includes:

- **Provisioning sufficient hardware** to run the job in a reasonable amount of wall clock time.
- **Batching requests intelligently** to combine as many inputs as possible into each forward pass.
- **Queueing requests and load balancing** across GPUs to ensure that requests are not lost and GPUs don't crash.
- **Recovering from any issues** that do occur and re-processing any missed requests.
- **Writing efficient and highly parallel client code** to ensure that the inference system is receiving as much traffic as it can handle.

These large offline workloads can take hundreds or thousands of GPU hours for extremely large corpora, so even modest throughput gains can yield huge cost savings.

CONTINUED >

Optimizing for online latency

In contrast, other embedding workloads must be optimized for online latency, where the user-facing time between request and response is essential. Throughput still matters, as you may need to support many concurrent users, but consistently low P90/P99 latency is key for online workloads.

Some applications where latency matters most include:

- **Multi-step AI agents** using embeddings as part of real-time actions
- **Real-time semantic search** in customer-facing apps
- **Live content recommendations**

Where latency-sensitive LLM requests are generally measured in the hundreds of milliseconds, embedding inference budgets are often just tens of milliseconds, including network latency. Achieving these instant responses requires:

- **Runtime optimizations** to use the most performant engines and kernels for inference.
- **Model quantization** to reduce load on the GPU during inference (done in floating point number formats to avoid loss in quality).
- **Autoscaling infrastructure** to keep latencies low when usage spikes.

At Baseten, we've seen all kinds of embedding workloads and developed tooling and expertise on handling everything from large-scale offline jobs to high-concurrency online requests. With the Baseten Inference Stack, we've developed the fastest and most scalable platform for running inference on open source, fine-tuned, and custom-built embedding models.

CONTINUED >

How Baseten built the fastest embedding inference stack

Solving for both latency and throughput requires end-to-end optimizations across the stack, from eliminating client code bottlenecks to building performant infrastructure to on-GPU model performance work.

At Baseten, our Inference Stack draws upon years of work on scalable, cloud-agnostic, fault-tolerant infrastructure and runtime optimizations. Together, four pieces of the stack deliver the world's fastest and highest-throughput embedding inference service:

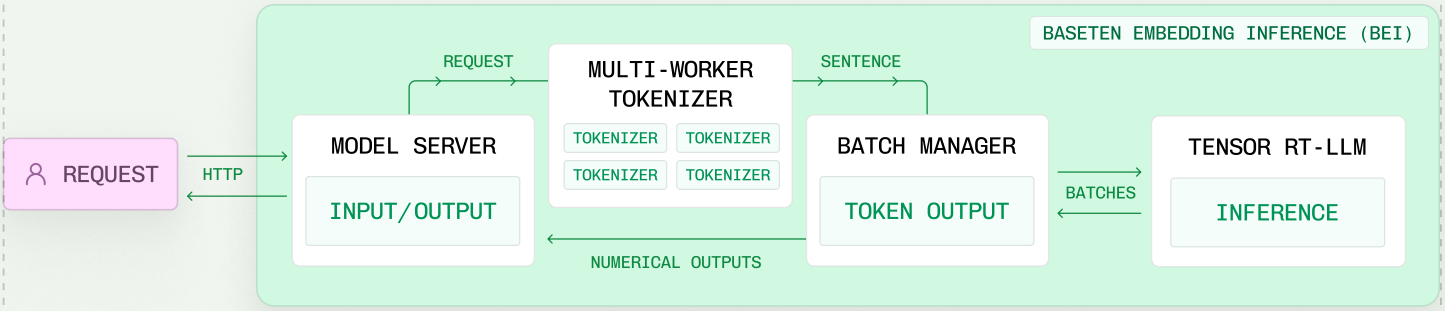
1. **Baseten Inference-optimized Infrastructure:** The core set of technologies powering cloud-agnostic and scalable infrastructure across all model deployments.
2. **Baseten Embeddings Inference (BEI):** Our best-in-class embedding-specific runtime with support for leading open source model architectures.
3. **Baseten Performance Client:** An open-source client library designed to avoid bottlenecks in running high-throughput embedding workloads.
4. **Baseten Chains:** Our framework for writing multi-step, multi-model compound AI pipelines and reducing latency overhead between steps.

With these tools, you can serve embeddings inference for any volume of traffic with incredibly tight latency budgets.

Baseten Embedding Inference (BEI)

Baseten Embedding Inference (BEI) is the world's most performant runtime for LLM-based embeddings models. In a benchmark, BEI on B200s achieved 3.3x higher throughput than vLLM and 3.6x higher throughput than TEI running on H100s.

BEI uses TensorRT-LLM's advanced kernels, FP8 quantization capabilities, and advanced batching with support for both Hopper and Blackwell GPU architectures to achieve the highest possible performance.



BEI has four main components:

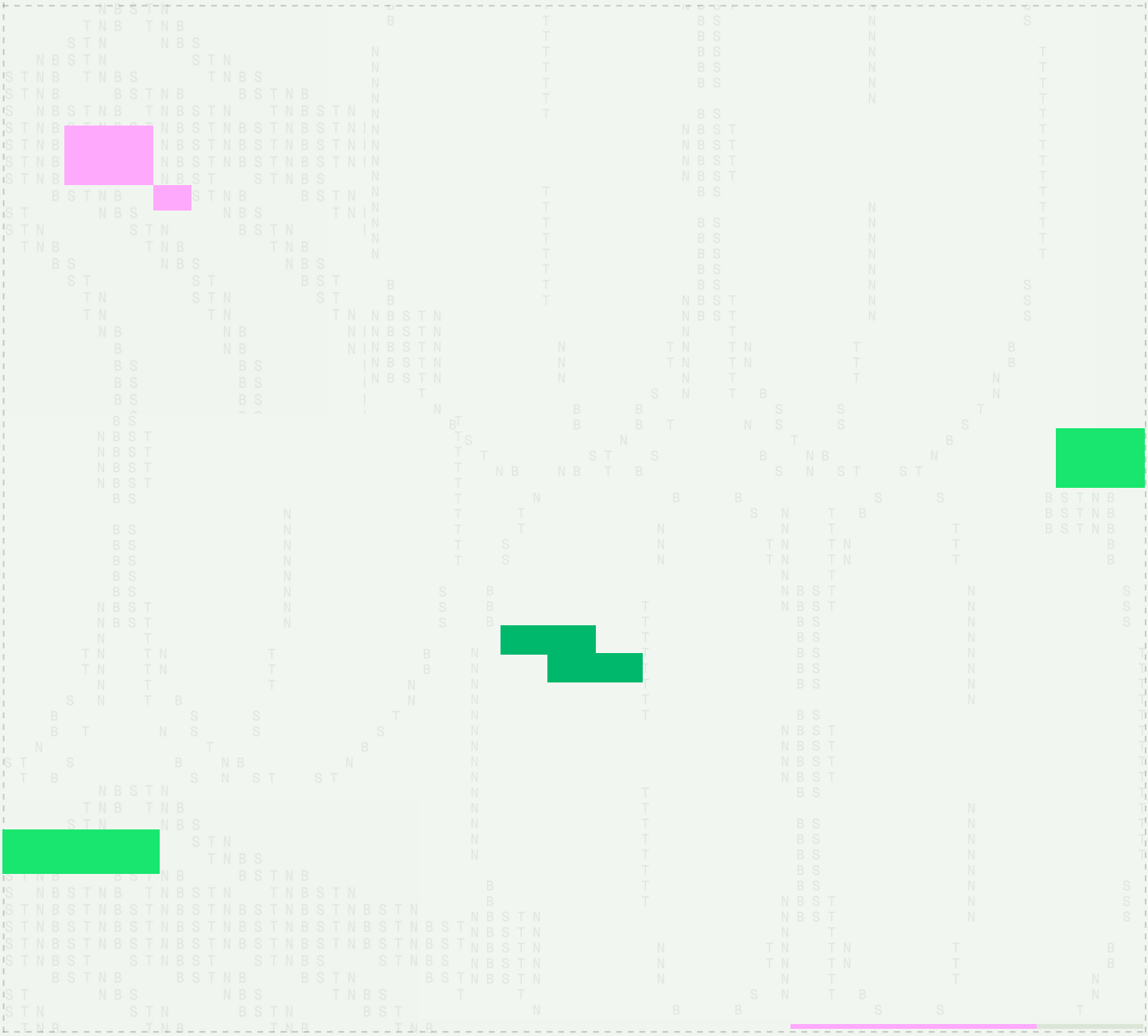
1. **The model server** processes inputs and outputs and handles any errors. BEI uses the Rust-based frontend service from text-embeddings-inference for this task.
2. **The tokenizer** is a multi-core system responsible for turning requests into tokenized sentences.
3. **The batch manager** packs individual tokenized sentences into batches up to a maximum sequence size, using a scheduling policy to maximize GPU utilization, minimize tail effects, and preserve request order.
4. **The TensorRT-LLM inference engine** runs inference in C++ using tokenized batches and creates embeddings.

Each request flows through all four components on the way in, but skips the tokenizer on the way out as embedding and classification outputs are numerical.

CONTINUED >

Baseten embeddings inference (BEI)

On the infrastructure side, BEI integrates seamlessly with Baseten's traffic-based autoscaling and async inference queueing, ensuring smooth handling of traffic spikes, efficient resource use, and graceful spin-down of replicas. This combination of throughput, concurrency, and low latency makes BEI a Pareto improvement for embedding, reranking, classification, and reward models.



Baseten Inference-optimized Infrastructure

At the infrastructure layer, embedding performance is generally based on three factors: hardware provisioning, geo-aware load balancing, and request queueing. In short, we need to ensure that there is a GPU ready to process the request, that the request goes to the right GPU, and that in high-throughput cases, the request can wait if the GPU is full.

Multi-cloud capacity management

Multi-cloud capacity management (MCM) is a set of automations, tools, and practices around provisioning and operating compute resources across multiple cloud service providers (CSPs) and regions in a standardized and repeatable manner.

Baseten's multi-cloud capacity management product:

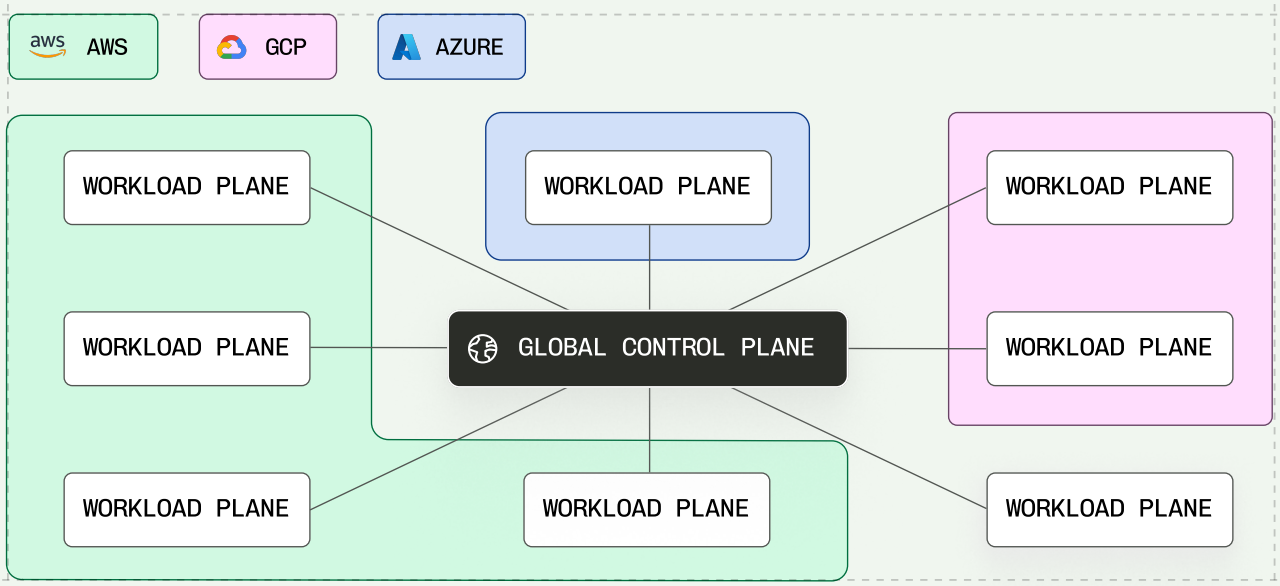
- Ensures high uptime with optional active-active reliability
- Supports the lowest possible latencies with multi-region deployments for geographic proximity to all end users
- Adheres to data residency and sovereignty requirements with region-locked deployments.
- Unlocks an optimal customer cost-performance ratio.

MCM makes siloed compute completely fungible: different clusters, regions, and cloud providers become one elastic, universal cloud. It's built on years of engineering work to make resource provisioning and allocation identical from provider to provider, despite the unique wrinkles that each CSP has, from networking stacks to exact resource SKUs.

At the core of MCM is a globally consistent orchestration layer built on top of Kubernetes with a global scheduler and a hub-and-spoke model.

CONTINUED >

Baseten inference-optimized infrastructure



We use MCM to route traffic across 10+ clouds and dozens of regions, optimizing for the closest resources while ensuring uptime. An H100 in us-east-1 on AWS becomes equivalent to an H100 in us-west4 on GCP.

Autoscaling with fast cold starts

Autoscaling is essential to systems like embeddings models with variable traffic. Autoscaling determines:

1. How quickly you can scale to handle a spike in traffic
2. How cost-efficient your hardware usage is when traffic slows back down

CONTINUED >

Baseten inference-optimized infrastructure

To keep latencies low as traffic scales, we need to dynamically allocate additional GPUs as the number of requests exceeds the batch size configured on the model server. For example, if you're serving Qwen3 Embedding on an H100 GPU with a batch size of 32 requests, but are now seeing 50 requests at a time coming in, you'd want to scale horizontally to a second GPU to avoid queue time increasing request latencies. Of course, when traffic settles back down, you'll want to automatically spin down that extra replica.

Baseten's Inference-optimized Infrastructure includes a traffic-based autoscaling system that holds requests in a queue while a new GPU is spun up, then routes those requests across the expanded profile of compute resources. A key component of this system is fast cold starts. New GPU resources need to be online in seconds to smoothly scale against sudden traffic spikes.

Large queue support

When processing a large corpus, we generally need to scale past a single replica. In a traffic-based autoscaling system, we might run in a steady state with a handful of replicas serving production traffic, then get hit with a huge corpus to process.

Baseten's Inference-optimized Infrastructure includes a queueing system to appropriately handle backpressure to keep everything running, enqueue additional requests as more replicas spin up (with fast cold starts), and distribute load evenly among the system as more replicas come online. Of course, the replicas will need to be gracefully spun down after the spike in usage.

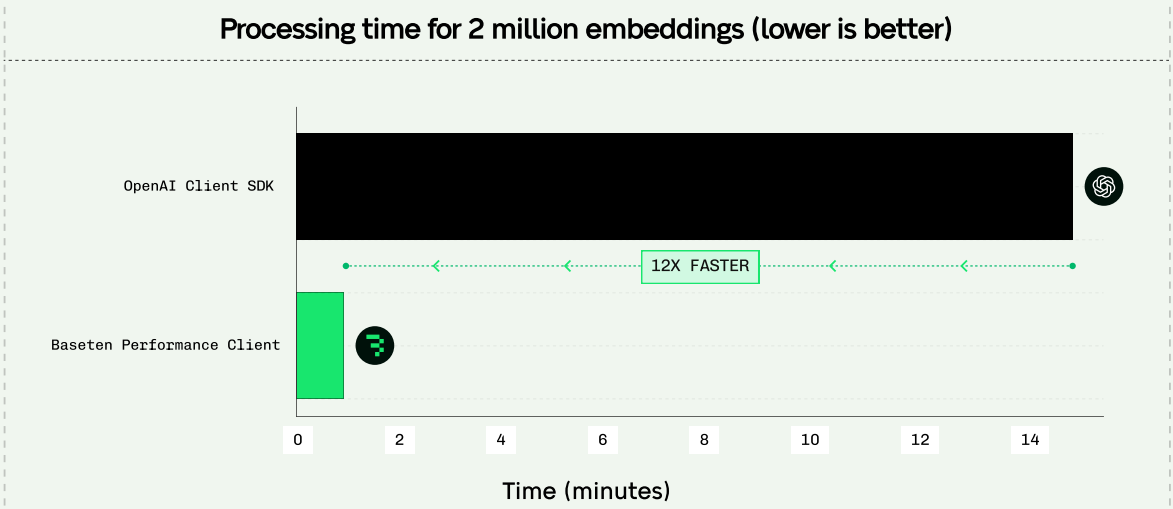
The infrastructure system also supports asynchronous inference for queue processing. With asynchronous inference, you get a response as soon as the request is enqueued. Once the inference output is ready, it's returned via webhook. This asynchronous setup is ideal for many high-volume offline workloads.

CONTINUED >

Baseten inference-optimized infrastructure

Baseten Performance Client

With a high-performance, high-throughput model server, client code can end up being the bottleneck in embedding system performance, especially in offline batch jobs. The Baseten Performance Client is a Python library written in Rust that overcomes these bottlenecks by properly parallelizing requests on the client side. The Baseten Performance Client is fully OpenAI-compatible and is a drop-in replacement for the OpenAI SDK, but delivers up to 12x better throughput for large batch embedding workloads.



Traditional Python clients are constrained by the Global Interpreter Lock (GIL), which serializes execution and limits CPU utilization in I/O-heavy scenarios. Even asynchronous approaches like asyncio cannot fully harness multi-core systems at scale. The Baseten Performance Client removes this bottleneck by using Rust and PyO3 to release the GIL during network-bound tasks. Embedding requests execute on a global Tokio runtime (a high-performance async executor in Rust), allowing true parallelism across CPU cores. The GIL is reacquired only to return results, minimizing Python overhead.

CONTINUED >

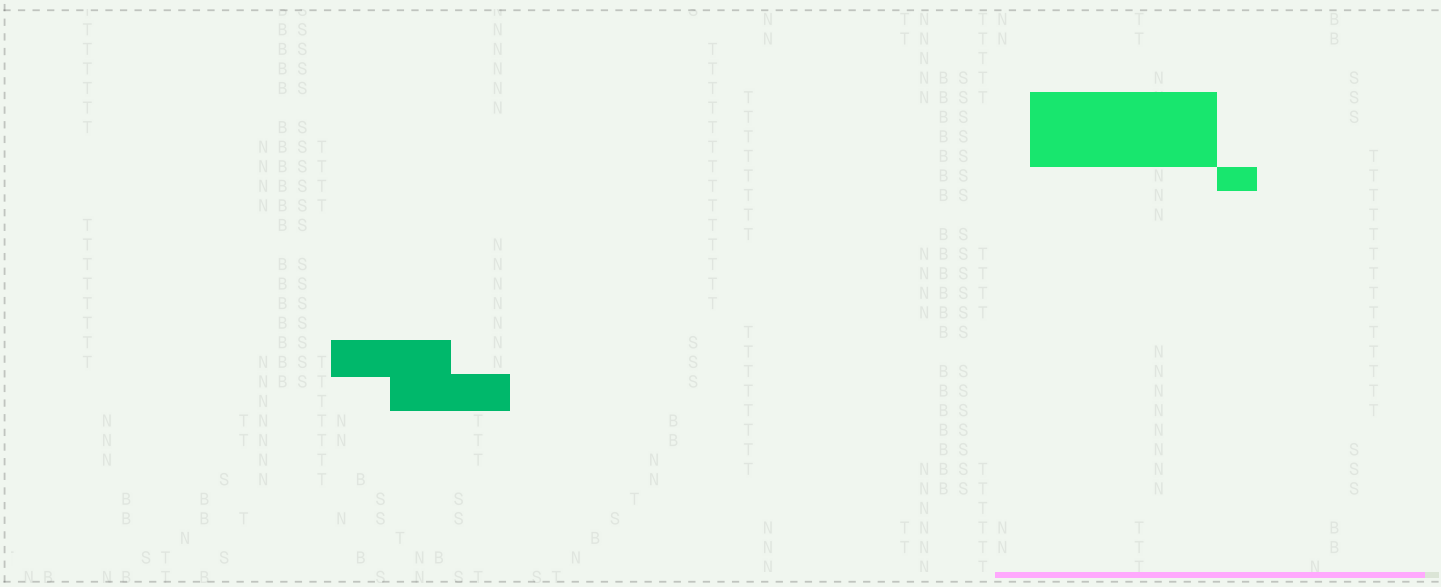
Baseten inference-optimized infrastructure

In benchmarks with over 2 million parallel inputs, the Performance Client completed embedding in 1:11 vs. 15+ minutes for the AsyncOpenAI client. It also achieved ≈280% CPU utilization on a 16-core machine, compared to 100% on a single core with Python clients. Using the Baseten Performance Client, you can ensure that every step of your embedding inference pipeline is fully optimized.

Baseten Chains

Many use cases for embedding models, like agentic memory and RAG, use an embedding model as part of a multi-step, multi-model pipeline. Baseten Chains is a framework for deploying these compound AI workloads on co-located infrastructure with independent component scaling, reducing network overhead in system-wide latency and preventing bottlenecks in autoscaling.

For example, an agent with a memory function that relies on embedding-based retrieval would ordinarily need to wait for the network overhead between the language model, business logic server, and embedding model service. With Chains, each of these components is co-located with appropriate resource allocation, saving dozens or even hundreds of milliseconds of latency and simplifying deployment.



Conclusion

Embedding models are no longer a niche capability, they are the foundation of modern AI applications, powering semantic search, RAG pipelines, personalization engines, recommender systems, and agent memory. But as their importance has grown, so too has the complexity of deploying them in production. The dual demands of high-throughput offline processing and low-latency online inference push traditional infrastructure to its limits.

Baseten's Inference Stack was purpose-built to meet these challenges. By combining cutting-edge runtime optimizations (BEI), cloud-agnostic multi-region infrastructure, intelligent autoscaling, and high-performance client libraries, Baseten enables organizations to serve embeddings at scale with unparalleled speed, reliability, and efficiency. Whether embedding billions of documents for indexing or delivering millisecond-level results in customer-facing applications, Baseten delivers the fastest runtime and the most cost-effective path to production.

As AI systems evolve toward more complex, multi-step agents and compound pipelines, embeddings will only become more critical. With Baseten, teams can confidently adopt open-source and fine-tuned models, unlock frontier-level performance, and future-proof their applications against platform risk. The result: scalable semantic infrastructure that keeps pace with the speed of innovation.

If you want to learn more about how to run embedding model inference at scale with Baseten, you can talk to our engineers.

Contributors

A big shout out and thank you to our contributors:

Philip Kiely

Michael Feil