Optimizing RAG Tail Latency

Refactoring Distributed Vector Search for High Scale Voice AI

CORE STACK: Postgres / pgvector | HNSW | Node.js | WebSocket | Redis

RESUME BULLET ANCHORED

> **RESUME ACHIEVEMENT**
> *"Reduced RAG p99 by 320 ms over 10 TB data by adding HNSW in pgvector and int8 scalar quantization in the retrieval path."*

SUMMARY

In real-time Voice AI, latency jitter is a primary cause of user abandonment. At Jambonz, RAG retrieval was a primary driver of Time to First Token (TTFT) in the LLM streaming path. I led the implementation to stabilize tail behavior by migrating our vector search from IVFFlat indexing to HNSW and implementing int8 scalar quantization in the retrieval path. These changes reduced p99 latency by 320 ms and stabilized p99 at 520 ms for our WebSocket-based streaming gateway.

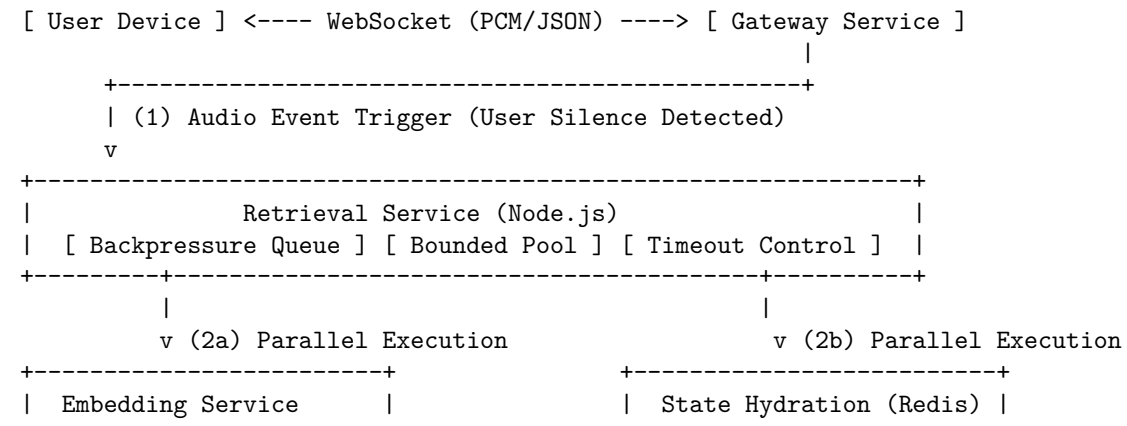
CONTEXT AND CONSTRAINTS

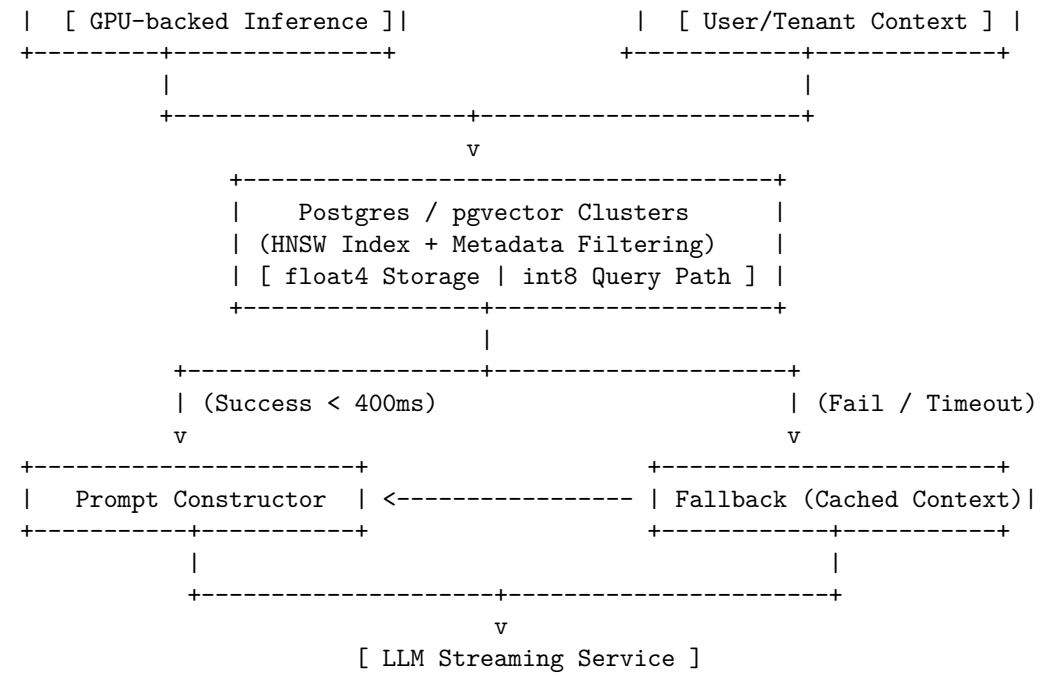
Voice AI requires a near-instantaneous response cycle to maintain natural conversation cadence. When a user stops speaking, the system must transcribe audio; retrieve context; and generate text within a tight budget to avoid audible dead air. At Jambonz, we managed a 10 TB source corpus across sharded Postgres clusters, supporting about 1.2M session records.

The constraints were threefold. First, we required metadata restricted search to ensure strict row level tenant isolation. Second, we needed to maximize memory efficiency to keep the working set in the OS page cache. Finally, the solution had to be implementable within our existing Postgres footprint to minimize operational risk and data synchronization complexity.

ARCHITECTURE

The diagram below details the optimized retrieval flow. Note the parallelization of the GPU-backed embedding generation and Redis-based session hydration phases to compress the critical path.





THE PROBLEM

Our legacy system utilized IVFFlat indexing. While median latency was acceptable, the p99 distribution frequently exceeded 800 ms due to three factors.

1. Planner bypass under selectivity. The Postgres query planner sometimes chose sequential scans when combining selective filters with nearest-neighbor ordering under IVFFlat cost estimates; this caused large p99 spikes in specific tenant environments.

2. I/O-bound retrieval. 1536 dimensional float32 vectors resulted in a massive index size. The working set frequently fell out of RAM, forcing slow disk fetches from cold storage that bottlenecked our p99 metrics.

3. Lack of Backpressure. A slow database shard could saturate shared pools and queues, increasing tail latency for otherwise healthy sessions sharing the same resources.

APPROACH AND TRADEOFFS

I evaluated external vector DB options and aligned with the team to stay on Postgres to reduce operational risk. We accepted longer index construction times for HNSW in exchange for more predictable tail latency and sublinear search behavior in practice. HNSW effectively turns the search space into a navigable graph; this reduced our search complexity from linear per bucket to a sublinear traversal.

IMPLEMENTATION NOTES

1. HNSW Indexing in pgvector. I migrated our sharded instances to HNSW and tuned the construction parameters to balance retrieval quality with index size. I adjusted the ef_search parameter based on the request type, prioritizing speed for simple turns while preserving recall for domain lookups.

2. int8 Scalar Quantization. To address the I/O bottleneck, I implemented a pipeline in the retrieval layer to convert float32 embeddings to int8. This reduced the vector footprint by 75 percent, allowing more of the index to reside in the OS page cache. This reduced p99 disk wait time from 400 ms to less than 45 ms.

3. Node.js Bounded Concurrency. I implemented bounded concurrency with a fixed queue depth around retrieval calls to introduce explicit backpressure. When a database shard hit saturation, the service served cached session context to avoid backlog growth.

VALIDATION

I built a query replay harness using 100k sanitized production queries. We used NDCG to confirm that ranking quality after int8 quantization changed by less than 0.5 percent. Load testing showed that the system maintained p99 stability at a sustained 500 requests per second per shard.

COMPARATIVE RESULTS

KPI Metric	Baseline (IVFFlat)	Optimized (HNSW + Q)
RAG p99 Latency	840 ms	**520 ms**
Disk I/O Wait (p99)	400 ms	**<45 ms**
Vector storage footprint	10.2 TB	**2.6 TB**
p99 SLO Compliance	94.2%	**99.8%**

MISTAKES AND COURSE CORRECTIONS

1. Maintenance Work Memory. Early index builds failed due to OOM errors. I resolved this by staggering maintenance windows and tuning maintenance_work_mem for index builds per shard to accommodate graph construction overhead.

2. Quantization Clipping. Initial int8 conversion caused precision loss for vectors with outlier scalar ranges. I corrected this by adding a normalization step and calibrating the clipping range based on the embedding model distribution.

KEY LEARNINGS

1. Tail Latency is Product Quality. In real-time systems, averages hide product failure. The p99 metric is the only meaningful signal for Voice AI stability.

2. Memory Locality is the Primary Lever. In 10 TB systems, algorithm speed matters less than the page cache hit rate. Quantization was the highest impact change in this project.

3. Backpressure is Safety. Bounded concurrency reduced risk during spikes by shedding load gracefully. Failing fast is a feature, not a bug.

CONFIDENTIALITY NOTE

All data has been sanitized. Revenue figures and proprietary customer identifiers have been omitted to maintain confidentiality while preserving technical accuracy.