Moorcheh Edge: On-Device Retrieval That Leaves Room for the LLM

A deep case study on edge RAG, MIB-based vector search, and the retail kiosk reference implementation on Arduino UNO Q

Version	1.0 (June 2026)
Platform	Arduino UNO Q (~3.6 GB RAM, ARM64 Linux)
Demo	The Brew Corner (fictional café and mini-market)
Product	Moorcheh Edge

Executive summary

Retail and field deployments need **grounded answers** from store-specific knowledge: menus, prices, policies, and FAQs. Cloud RAG (retrieval-augmented generation) works in the data center but breaks down for kiosks that require **privacy**, **offline resilience**, and **predictable cost**. Running the same stack on a small edge board is harder than it looks: the embedding model, vector index, and LLM each consume hundreds of megabytes to gigabytes of RAM.

Moorcheh Edge targets the retrieval layer. Float embeddings are **compressed** with **MIB (Maximum Information Binarization)** into compact one-bit codes. At query time, **EDM (Efficient Distance Metric)** scores similarity directly on those codes. Only quantized bytes are persisted; the search server stays small.

This architecture is described in Moorcheh's research paper *From HNSW to Information-Theoretic Binarization*, which replaces the conventional **HNSW + float32 + cosine similarity** stack with an information-theoretic approach suited to low-RAM and edge devices.

On Arduino UNO Q we measured:

Result	Value
Maximum store capacity	**10,000** vectors (768-d)
Search latency at capacity	**~17 ms** median (top_k=5)
Store data on disk	**2.0 MiB**
Moorcheh Docker container RAM	**~20 MiB** total (~2 MiB data + ~18 MiB runtime)
Float32 equivalent vector payload	**~31 MiB** (not used by Moorcheh)

We prove the stack in a **retail kiosk demo**: Moorcheh **client** on a display PC; **BGE embedding**, Moorcheh **server**, Ollama, and voice on UNO Q.

This document explains why naive edge RAG fails on RAM, how Moorcheh differs, what we built, and what else fits on the same hardware.

Table of contents

---

1. Why edge AI for in-store assistants

1.1 The in-store context

A customer at a shelf is not using a SaaS dashboard. They speak or tap briefly, expect an answer in seconds, and move on. The assistant must know **this store's** prices, hours, allergens, and promotions - not generic web knowledge.

Cloud assistants send every question (and often audio) to remote APIs. That creates:

- **Latency** - round-trips for embed, search, and generate add hundreds of milliseconds to seconds.
- **Privacy** - questions about health, diet, or purchases leave the premises.
- **Reliability** - spotty Wi-Fi should not silence the kiosk.
- **Cost** - per-token cloud LLM fees scale with foot traffic.

Edge deployment keeps retrieval and generation **on device** after content sync. Moorcheh Edge is the **retrieval** piece; a small local LLM is the **generation** piece.

1.2 Why retrieval is the bottleneck on small boards

Teams often focus on shrinking the LLM (1B parameters, Q4 quantization). That is necessary but not sufficient. A typical RAG stack also needs:

1. An **embedding model** to turn text into vectors (~200 MB+ on disk, hundreds of MB RAM at inference).
2. A **vector store** holding thousands of document chunks (float indexes can be tens of MB in RAM alone).
3. The **LLM** itself (often 800 MB-2 GB+ peak RAM for small models).
4. Optional **voice** (STT + TTS) adding more RAM and CPU.

On a **~3.6 GB** board, every megabyte matters. Moorcheh's bet: make (2) tiny so (1), (3), and (4) can coexist.

1.3 Reference platform: Arduino UNO Q

Spec	Value
SoC	Qualcomm ARM64 (Linux application processor)
RAM	~3.6 GB usable (4 GB model)
OS	Debian Linux
Containers	Docker
Role in demo	Runs Moorcheh Edge, Ollama, voice server

The UNO Q is our **reference edge MPU** for the kiosk. All AI workloads in this demo run on the Linux side; MCU features are reserved for future sensor/UI integration.

---

2. What RAG is and why it matters here

2.1 Retrieval-augmented generation

RAG combines:

1. **Retrieval** - find the most relevant passages from a local knowledge base.
2. **Augmentation** - inject those passages into the LLM prompt as context.
3. **Generation** - the LLM produces an answer grounded in that context.

Without retrieval, a small on-device LLM hallucinates prices and policies. With retrieval, answers cite (implicitly) synced store content.

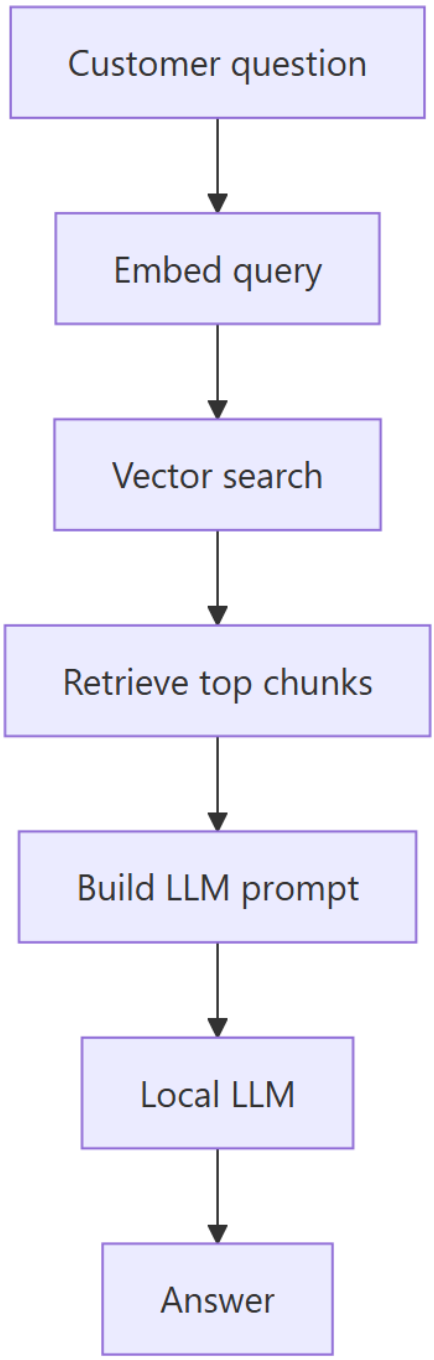

Figure 1: Retrieval-augmented generation pipeline from question to grounded answer.

2.2 Pipeline stages

```
Customer question
   → embed query (768-d float vector)
   → search vector store (top-k chunks)
   → build prompt: system + chunks + question
   → LLM stream answer
   → (optional) TTS to speaker
```

Seven figures are embedded in the PDF (RAG pipeline, edge comparison, MIB/EDM, RAM budget, kiosk architecture, upload path).

2.3 Two store modes in Moorcheh Edge

Mode	Who embeds	Dimension	Typical use
text	Python client (BGE-base-en-v1.5)	768 (fixed)	Kiosk catalogs, FAQs
vector	Your app (precomputed)	128-1536 (allowlist)	Benchmarks, custom embedders

Mode is locked on first upload. The retail kiosk uses **text** mode; our capacity benchmark uses **vector** mode (random vectors, no chunk text).

---

3. Why running full RAG on the edge is hard

3.1 The naive approach

"Run everything on the board" usually means:

- Download an embedding model to the device.
- Build a **float** vector index (cosine similarity or HNSW) in RAM.
- Run a quantized LLM in Ollama.
- Optionally run STT/TTS.

Each layer is defensible alone. Together on 3.6 GB they compete for the same RAM.

3.2 RAM math (order of magnitude)

Layer	Typical pressure on UNO Q
Embedding inference (BGE-base)	Hundreds of MB in Python/ONNX process
Float vector index (10k × 768 float32)	**~31 MB** raw vectors only; **plus** HNSW/graph index overhead (often tens of MB more); **plus** separate DB runtime process
LLM (1B class, Q4)	~800 MB weights; **~1 GB+** peak RSS observed (`llama-server`)
Moorcheh MIB store data (10k vectors)	**~2 MiB** quantized binary payload (measured on disk and in RAM)
Moorcheh Edge container (total)	**~20 MiB** at 10k vectors - **server, store, and search in one container** (measured)
Voice (Whisper + Piper)	Additional RAM; often sequential with LLM

Float index vs Moorcheh on the same 10k corpus:

	Float ANN-style stack	Moorcheh Edge
Vector payload	~31 MB (float32)	**~2 MiB** (MIB codes)
Index overhead	HNSW graph, metadata, DB buffers (often **+50-200+ MB** depending on implementation)	**None** - no ANN graph; store is the index
Runtime	Vector DB process + your app	**One ~20 MiB container** (Rust server + HTTP + loaded store)

Observed top processes during development:

Process	RSS
`llama-server`	~1,022 MiB
`python3` (embed daemon)	~676 MiB
`moorcheh-edge` (Docker)	~20 MiB

Takeaway: The LLM and embedder dominate. A float vector stack adds payload **and** index **and** DB runtime. Moorcheh keeps retrieval in a **single small container** (~2 MiB data, ~20 MiB total at 10k).

3.3 Why cloud-style vector databases do not map cleanly to edge

Managed and self-hosted **vector databases** (generic similarity search services, pgvector-style extensions, embedded HNSW libraries) optimize for datacenter scale: many collections, float precision, hybrid filters, replication. On edge they imply:

- Large in-memory graph or float tables.
- Operational complexity (sidecars, sync agents).
- Unpredictable RAM growth as corpus grows.

HNSW is rarely run on the device. Most products that use HNSW keep indexing and search on a **remote server** (managed vector DB or cloud API). Building and holding an HNSW graph in RAM on a 3.6 GB board alongside an embedder and LLM is usually impractical: index construction is heavy, the graph must stay resident for low latency, and RAM use grows with corpus size.

Moorcheh Edge takes the opposite path: **no HNSW graph**, MIB-quantized store, EDM scan in a **~20 MiB container** so retrieval can stay on UNO Q with the LLM. It deliberately caps at **one flat store, 10k items** - not feature parity with cloud vector SaaS, but a fit for fixed-location devices.

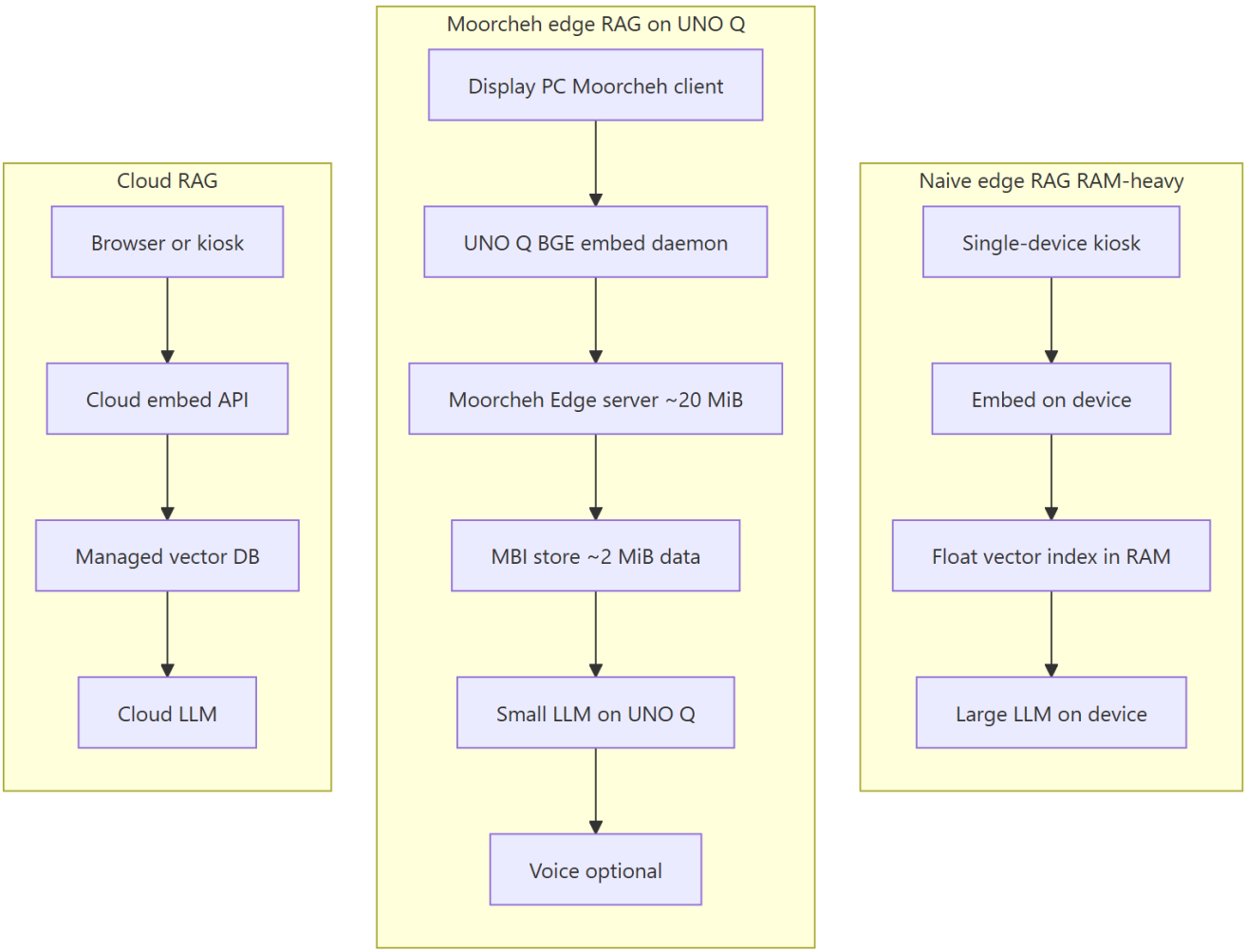

Figure 2: Three deployment patterns. Moorcheh keeps retrieval small so a local LLM and voice can fit on UNO Q.

3.5 Sequential vs concurrent workloads

On 3.6 GB, running **Whisper + warm LLM + embed daemon + Moorcheh** concurrently can spike RAM. Production kiosks may:

- Pre-warm LLM with `--warm-llm` for low first-token latency.
- Run voice **sequentially**: record → STT → RAG → TTS, not parallel STT+LLM on largest models.
- Stop embed daemon during voice-only CLI tests if RAM is tight.

This is operational tuning, not a Moorcheh limitation - but teams must plan for it when marketing "full voice RAG on edge."

3.6 When cloud RAG still wins

Be honest in customer conversations. Cloud RAG remains appropriate when:

- Corpus exceeds **millions** of chunks or needs hybrid SQL filters.
- You require the largest frontier models without quantization.

- Latency to a regional API is acceptable and connectivity is reliable.
- Per-query cost is negligible at your volume.

Moorcheh Edge targets the **long tail of fixed-location devices** with bounded knowledge (one store, one clinic FAQ set, one manual) - not general web search.

---

4. Moorcheh Edge: MIB vector compression and EDM retrieval

4.1 How Moorcheh differs from conventional vector search

Most vector databases use the same three-part stack (arxiv:2601.11557):

Conventional stack	Moorcheh stack
HNSW graph index in RAM	**No ANN graph** - exhaustive scan over compact codes
float32 vectors (~3 KB per 768-d vector)	**MIB** one-bit codes (~96 bytes per 768-d vector)
Cosine similarity on floats	**EDM** on quantized codes

That swap matters on edge hardware. A billion 768-d float32 vectors need on the order of **terabytes of RAM** for in-memory ANN. Moorcheh's paper shows MIB can preserve semantic signal in **~32x less memory**, enabling exhaustive search on binary representations at CPU-friendly cost.

Moorcheh Edge applies this stack on-device: small store, deterministic scan, no cloud vector DB.

4.2 Design goal on UNO Q

Keep the **search server** small and fast on ARM64. The client sends float embeddings (or text that the embed daemon converts); the server applies **MIB**, stores quantized codes, and retrieves with **EDM** - all without cloud calls.

4.3 MIB - Maximum Information Binarization

MIB is a loss-minimizing **vector compression** step that converts a dense float embedding into a compact, information-preserving **one-bit-per-dimension** code.

Unlike naive binarization (for example random-projection LSH), MIB is designed to keep the **most information-rich dimensions** of the original vector in the binary form. On Moorcheh Edge, each uploaded vector becomes a packed byte array (`binarized_embedding`); the original floats are **not** kept on disk.

For 768 dimensions, storage is **96 bytes** per item. At 10,000 items, the vector payload alone is about **0.92 MiB** in RAM.

4.4 EDM - Efficient Distance Metric

At query time, the query vector is binarized with the same **MIB** process. **EDM (Efficient Distance Metric)** measures similarity between the query code and each stored code.

EDM is built for quantized representations: it uses fast, CPU-native bitwise operations instead of high-dimensional float cosine math. That is why Moorcheh Edge can scan **10,000 vectors in ~17 ms** on UNO Q while the Docker container stays near **~20 MiB** total RAM.

Results below a configurable **threshold** are filtered; **kiosk_mode** applies stricter filtering for voice UX.

4.5 Research reference

For the full architectural argument, benchmarks across 14 datasets, and comparison to Elasticsearch, Pinecone, PGVector, and Qdrant, see:

From HNSW to Information-Theoretic Binarization: Rethinking the Architecture of Scalable Vector Search (2026)

Key ideas from that work relevant to edge:

- **Geometric search is not the only path** - information-theoretic encoding can match or beat float cosine on retrieval quality in evaluated benchmarks.
- **Compression and speed are coupled** - MIB + EDM make exhaustive scan practical, so you do not need a large HNSW graph in RAM.
- **Serverless and edge share the same win** - compact codes and cheap comparison suit both Lambda-style functions and boards like UNO Q.

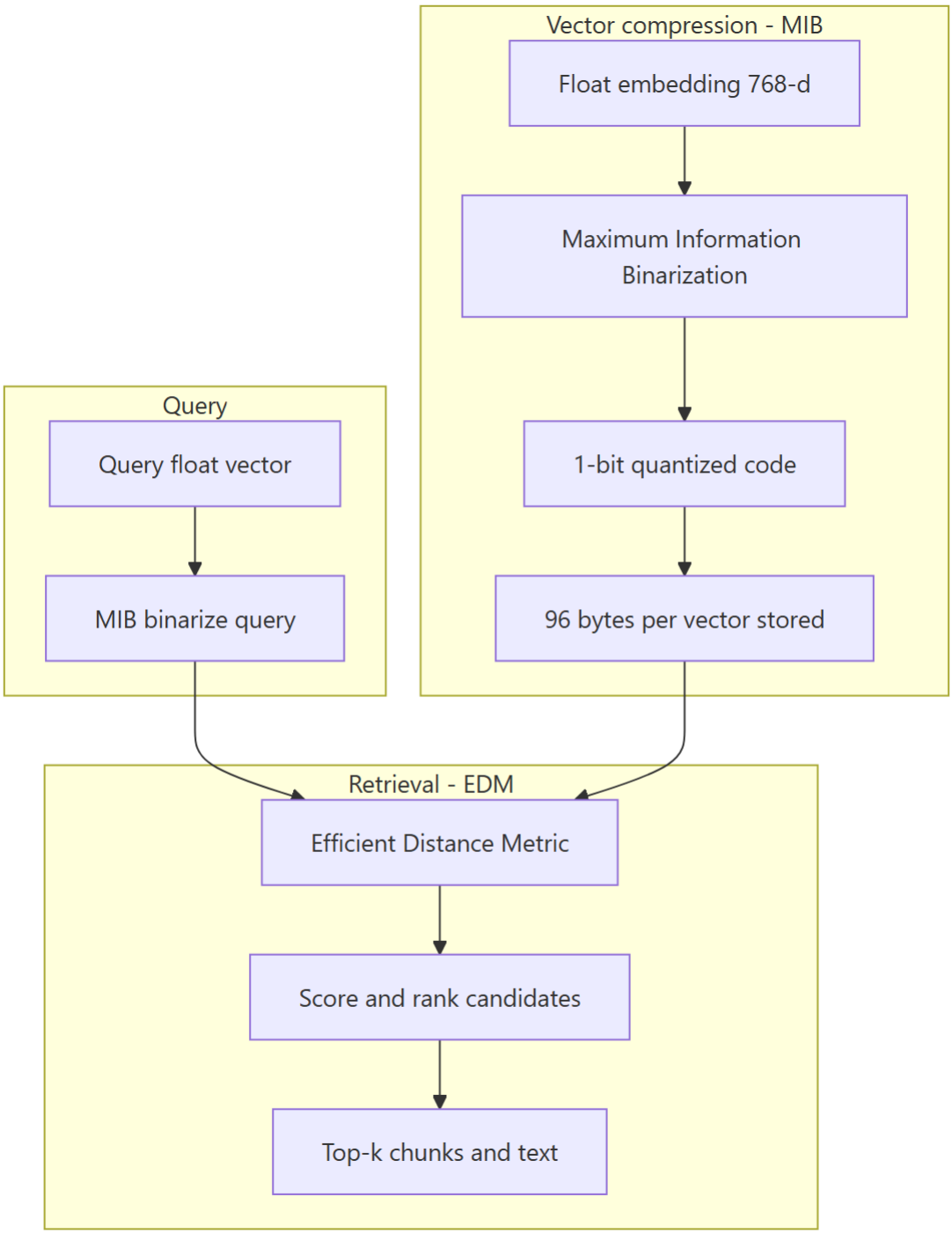

Figure 3: Float vectors are compressed at upload with MIB. Queries use the same compression; EDM scores and ranks stored codes.

4.6 What lives in RAM

The full store is loaded into a `HashMap` at server start and kept for linear scan search. For 10k items:

Component	Size
Binarized payloads	0.92 MiB
IDs + structures	~0.7-1.0 MiB

Component	Size
Data subtotal	**~2 MiB**
Rust runtime + HTTP	~18 MiB
Container total	**~20 MiB**

4.7 Comparison at 10k × 768-d

Approach	Vector payload RAM	Search at 10k (measured)
Float32 in-memory table	~31 MiB	Depends on index
HNSW + float32 (typical cloud/edge DB)	Graph + floats in RAM	Low ms but RAM-heavy
Moorcheh MIB + EDM	**~0.92 MiB** (+ text if stored)	**~17 ms** median

4.8 What Moorcheh does not do

- Replace the LLM or embedding model.
- Guarantee chatGPT-class open-domain quality.
- Support unlimited corpus size (10k cap per device).
- Run cloud-scale hybrid search or SQL filters.

It **does** deliver fast, grounded retrieval for store FAQ and catalog workloads on constrained hardware.

---

5. Measured results on Arduino UNO Q

All figures below were measured on **Arduino UNO Q** with Moorcheh Edge running in Docker, June 2026.

5.1 Test conditions

- **Store load:** 10,000 random unit vectors, 768 dimensions, **vector mode** (no chunk text)
- **Search:** top 5 results per query, raw query vectors (no client-side embedding during search)
- **Platform:** ~3.6 GB RAM, ARM64 Linux

5.2 Upload performance

Metric	Value
Upload time	26.751 s
Throughput	373.8 vectors/s

Metric	Value
Final store	`items=10000`, `dimension=768`, `store_mode=vector`

5.3 Search latency

Stat	ms
min	16.00
median	**17.16**
mean	17.29
max	18.67
stdev	0.87

Sub-20 ms search at maximum capacity leaves budget for embed + LLM + voice in end-to-end flows.

5.4 Memory and disk

Metric	Value
`moorcheh_edge_store.json`	**2.0 MiB**
Docker `MEM USAGE`	**~20 MiB**
Store data in RAM (estimate)	**~2 MiB**

Note on `clear-store`: Clearing items did not reduce Docker memory (~20.5 MiB after clear). Rust retains freed heap; use **container restart** to measure empty baseline. Disk file size is the most honest proxy for serialized store size.

5.5 Interpreting results for product planning

- **Retrieval is solved at 10k** on this board with headroom.
- **End-to-end latency** is dominated by embed (if on device), LLM tokens, and voice - not search.
- **Full stack RAM** still requires careful model choice (BGE-base + 1B Q4 LLM + sequential voice).

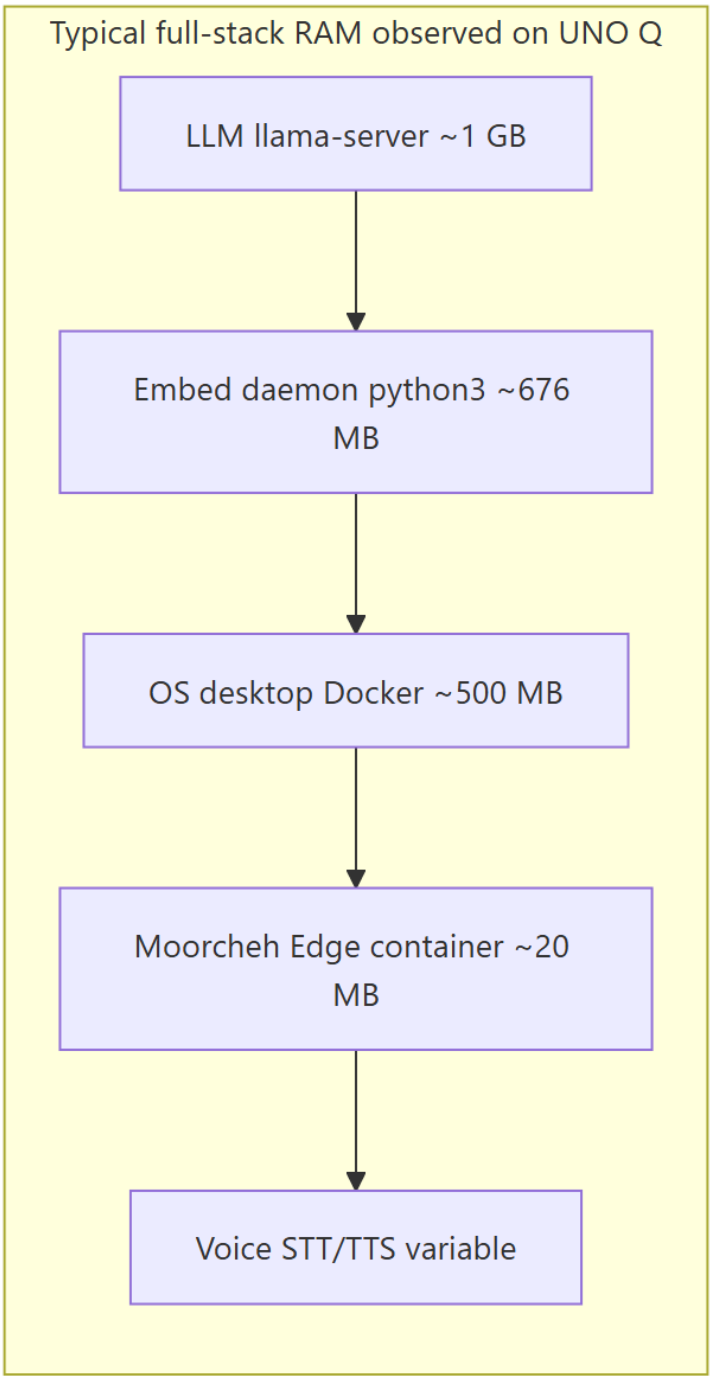

Figure 4: Observed RAM consumers on a 3.6 GB board. Moorcheh retrieval is the smallest layer.

---

6. Architecture pattern: split stack, not monolith

6.1 Recommended edge RAG topology

```
┌──────────────────── Display PC (Moorcheh client) ────────────────────┐
│  React UI  →  retail-kiosk-api  →  SQLite catalog metadata           │
└──────────────────────────┬────────────────────────────────────────────┘
```

```
                          | text, chunks, SSE
┌─────────────────────────▼──────────────────────────────────────────┐
│                  Arduino UNO Q (edge compute + server)             │
│  BGE embed daemon :18765  →  Moorcheh Edge server :8080            │
│  Ollama :11434  →  voice serve :8766  →  mic + speaker             │
└────────────────────────────────────────────────────────────────────┘
```

6.2 Responsibilities

Component	Location	Why
UX, Admin, catalog DB	PC	Rich UI, fast iteration
Moorcheh **client** (API)	PC	Sends text to UNO Q; no search server on PC
BGE embedding	UNO Q	`moorcheh-edge up` starts embed daemon on device
Vector store + search (MIB + EDM)	UNO Q	Moorcheh Edge **server** in Docker
LLM	UNO Q	Local answers, no cloud tokens
Voice I/O	UNO Q	Mic and speaker attached to board

The Python client on the PC calls the **embed daemon and Moorcheh server on UNO Q**. Embedding computation (BGE inference) runs on the board, not on the display PC.

6.3 Deployment note

Some tight-RAM demos may colocate embed on a gateway PC; the reference Moorcheh Edge shape keeps **embed + server + LLM on UNO Q**. Voice (`voice serve`) always embeds locally on the board before RAG.

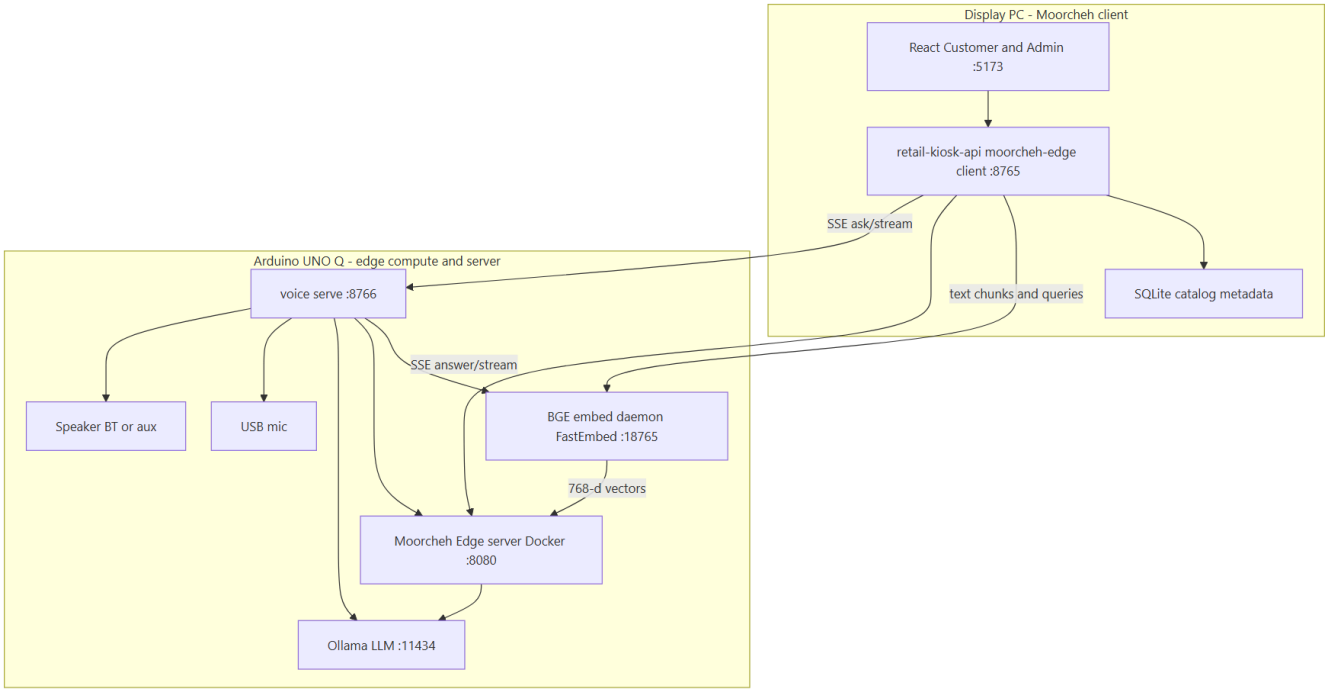

Figure 5: Display PC runs the Moorcheh client (UI + API). UNO Q runs BGE embedding, Moorcheh server, LLM, and voice.

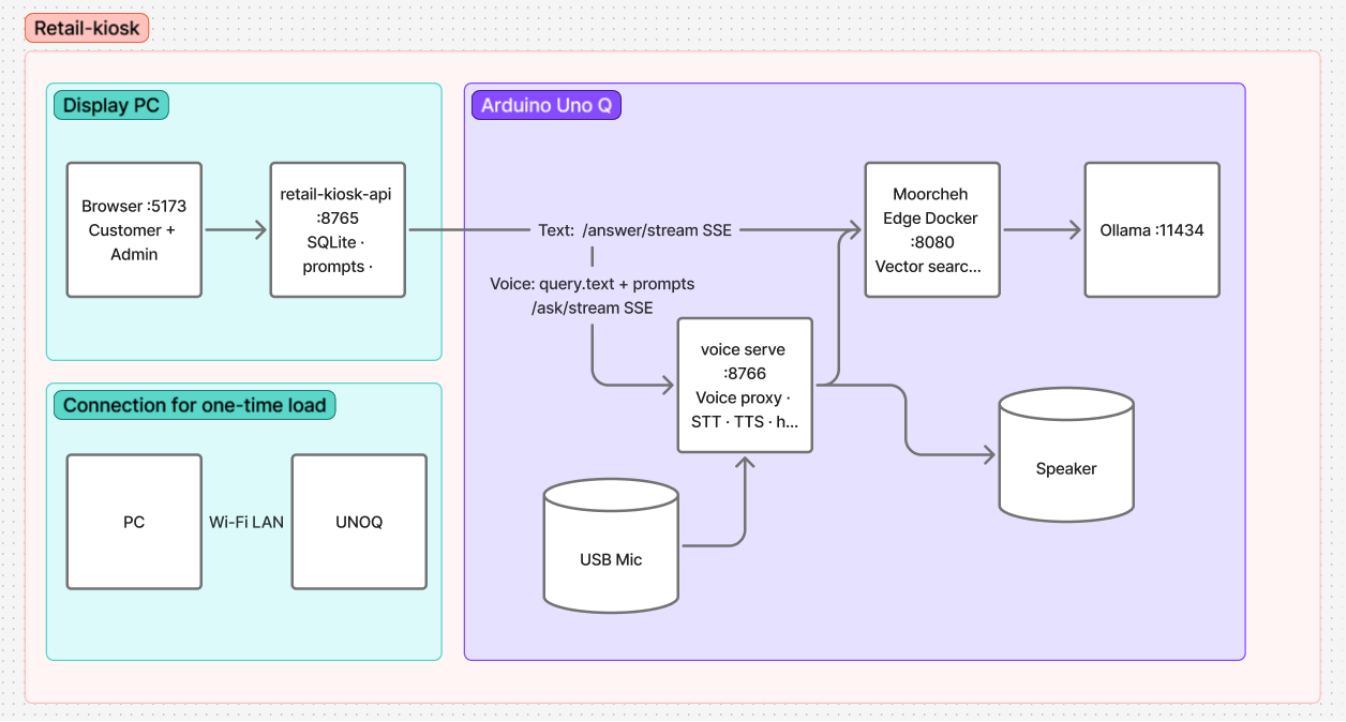

Figure 6: Component diagram from the retail-kiosk demo (React UI, API, Moorcheh Edge, Ollama, voice).

---

7. Case study: The Brew Corner retail kiosk

7.1 Scenario

The Brew Corner is a **fictional** neighborhood café and mini-market used as demo content. It sells coffee, pastries, grocery staples, and policies typical of a hybrid retail site. The kiosk answers questions like:

- "How much is a latte with oat milk?"
- "Do you have gluten-free options?"
- "What's your return policy on unopened groceries?"

7.2 Hardware

Piece	Detail
Display PC	Windows/Linux/macOS; React + Python API
Arduino UNO Q	Moorcheh Docker, Ollama, voice
USB microphone	Capture on UNO Q (`plughw:0,0` in voice config)
Speaker	Bluetooth (e.g. Anker Soundcore 2) or 3.5 mm aux

Piece	Detail
Wiring guide	Figma board

7.3 Software stack

Port	Host	Service
5173	PC	React UI (Customer `/`, Admin `/admin`)
8765	PC	`retail-kiosk-api` (Moorcheh client)
18765	UNO Q	BGE embed daemon (`moorcheh-edge up`)
8080	UNO Q	Moorcheh Edge **server**
11434	UNO Q	Ollama
8766	UNO Q	`moorcheh-edge voice serve`

See Figure 6 for the component diagram.

7.4 User journeys

Text path

1. Customer types a question in the UI.
2. PC API sends text to UNO Q; **BGE embed daemon** on the board vectorizes the query.
3. Moorcheh Edge searches the store and streams LLM tokens via `POST /answer/stream` (SSE).
4. UI renders streaming text.

Voice path

1. Customer presses talk; audio streams to `voice serve` on UNO Q.
2. Whisper STT → text question.
3. **Embed daemon on UNO Q** vectorizes the query; RAG runs on `:8080` → Ollama → Piper TTS → speaker.

Voice holding messages can play while RAG runs (`voice.holding_enabled` in catalog).

7.5 Admin workflow

- **Documents** - create/edit/delete knowledge; each save chunks text on the PC, sends chunks to UNO Q for **edge embedding**, then syncs vectors to Moorcheh Edge server.
- **LLM prompts** - header (system) and footer prompts for every answer.
- **Chunk catalog** - read-only view of chunk IDs on edge.

First-time demo load uses the Brew Corner catalog JSON (~27 documents). Ongoing edits use the Admin UI.

7.6 LLM and prompts

Default model: `qwen2.5:0.5b-instruct` via Ollama. Prompts instruct the assistant to answer **only from context**, speak naturally for voice, and omit address/phone (customer is already in store). When no passages match, the server returns a fixed no-information message without calling the LLM.

7.7 Voice configuration highlights

Setting	Purpose
`~/.moorcheh-edge/voice/audio.json`	USB mic capture; `"playback": "bluetooth"` or ALSA device
`MOORCHEH_BT_DEVICE`	Optional MAC filter for Bluetooth sink
`voice ask --kiosk-mode --threshold 3.0 --top-k 2`	Stricter retrieval + fewer chunks for spoken answers
`voice serve --port 8766`	HTTP SSE proxy for kiosk UI

Kiosk mode maps to server-side `kiosk_mode` on search: low EDM scores are dropped so the LLM does not read irrelevant chunks aloud.

7.8 Content scope of the demo catalog

The Brew Corner JSON includes **27 documents** across categories: store info, menus, grocery, policies, loyalty, allergens, checkout, promotions, seasonal items, catering, community board, and more. This breadth lets demo questions span price lookup, policy Q&A, and small-talk boundaries without live CMS integration.

---

8. What gets uploaded to the vector store

8.1 Catalog source

The Brew Corner demo catalog (`brew-corner-catalog.json`) contains:

- `store` - name and tagline
- `prompts` - header/footer for LLM
- `voice` - holding audio settings
- `documents[]` - `doc_id` , `category` , `title` , `tags` , `text`

8.2 Chunking and upload pipeline

For each document:

1. Split body into chunks of **≤200 tokens** (BGE limit 512; kiosk uses conservative chunks).

2. Prepend a **[meta]** header:

```
[meta]
doc_id: coffee-menu
category: menu
title: Coffee and espresso menu
tags: coffee,espresso,latte,prices,menu
chunk: 0
[/meta]

<chunk text>
```

3. Embed each full chunk with **BGE-base-en-v1.5** on **UNO Q** (embed daemon started by `moorcheh-edge up`).
4. Upload batch:

```
{
  "store_mode": "text",
  "embedding_model": "BAAI/bge-small-en-v1.5",
  "items": [
    { "id": "coffee-menu#chunk-0", "text": "...", "vector": [768 floats] }
  ]
}
```

5. Server applies **MIB**, stores quantized codes + text, persists JSON under `~/.moorcheh-edge/data/`.

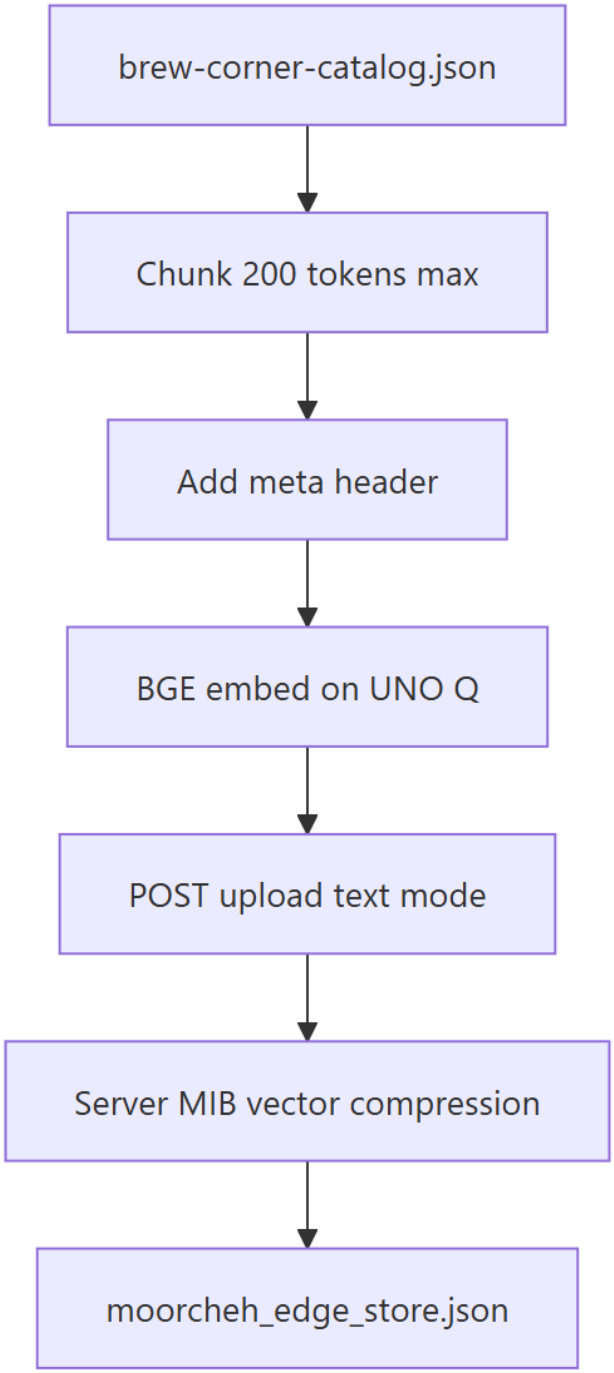

Figure 7: From catalog JSON to MIB-compressed vectors on the edge store.

---

9. Challenges and lessons learned

9.1 ARM64 Docker on Windows dev loop

UNO Q requires `linux/arm64` Docker images. Images are built for ARM64, exported, and transferred to the board for local install.

9.2 Ollama on UNO Q

Official installer URL issues were fixed by pulling `ollama-linux-arm64.tar.zst` into `~/.moorcheh-edge/ollama/`. `OLLAMA_HOST=0.0.0.0:11434` lets the Docker container reach Ollama via `host.docker.internal`.

9.3 Embedding model selection

We tried higher-scoring embed models (e.g. Arctic) but **cosine benchmarks did not predict** behavior after **MIB + EDM**. Irrelevant queries still matched too often. **BGE-base-en-v1.5** remained the practical default for English kiosk content.

9.4 Bluetooth audio on UNO Q

PipeWire Bluetooth sinks may live in the **lightdm** session while Moorcheh runs over SSH as **arduino**. Native `"playback": "bluetooth"` in voice config finds the sink and can play via `sudo -u lightdm` when needed.

9.5 RAM measurement literacy

- `docker stats` shows container total, not just store.
- `clear-store` does not shrink RSS.
- Compare **file size** and **fresh restart** for data-only estimates.

9.6 Namespace complexity removed

Early multi-namespace API parity with Moorcheh On-Prem was removed. Kiosk model: **one store per device**, simpler ops.

---

10. What else you can build on this hardware

With **~2 MiB** retrieval data at 10k vectors and **~17 ms** search, the same UNO Q stack supports other verticals by swapping catalog content:

Vertical	Example questions	Edge benefit
Pharmacy kiosk	Hours, pickup policy, OTC aisle FAQ	Privacy
Museum / gallery	Exhibit descriptions, tour times	Offline
Factory floor	SOPs, safety checklists	No cloud dependency
Clinic waiting room	General FAQs (non-diagnosis)	Local processing
Smart home hub	Device manuals, routines	Low latency

Vertical	Example questions	Edge benefit
Automotive (future)	Owner manual sections	In-vehicle

10.1 Scaling content

- **Per device:** up to 10k chunks; sync updates from a central CMS or Moorcheh On-Prem (roadmap).
- **Per fleet:** many UNO Q boards, each with a local store; PC or cloud pushes document updates.

10.2 When to keep embed on PC vs move to device

Embed on PC	Embed on UNO Q
Faster queries	No PC required
Lower board RAM during admin	Simpler wiring
Demo default	Air-gapped retail

---

11. Limits and honest tradeoffs

Limit	Detail
10k items	Hard cap per container
English BGE	Text mode locked to 768-d BGE-base
Linear scan	Fast at 10k; not designed for millions of vectors
LLM quality	Small model; tuned prompts essential
Concurrent voice + LLM	RAM may require sequential stages
10k + warm LLM + embed on same board	Documented as tight on 3.6 GB; split stack recommended

Moorcheh Edge is **not** a drop-in replacement for datacenter vector databases. It is a **retrieval engine for one device** where RAM and milliseconds matter.

11.1 Roadmap themes (not commitments)

Theme	Direction
On-Prem sync	Push document updates from Moorcheh On-Prem to edge stores
Richer voice UX	Holding audio, promo strings from catalog
Quantized embed	Lower RAM if BGE-base still tight on 2 GB-class boards

Theme	Direction
MCU integration	UNO Q microcontroller for physical UI later

---

12. Conclusion and next steps

Edge RAG fails when teams copy cloud architecture onto a 3.6 GB board. Moorcheh Edge shows a different path: **MIB** compresses vectors at upload; **EDM** retrieves at query time; the measured store holds **10k vectors in 2 MiB** with **~17 ms** search and a **~20 MiB** server footprint.

The **Brew Corner** retail kiosk demonstrates the full pattern - PC for UX and Moorcheh **client**, UNO Q for **embedding**, search, LLM, and voice - using fictional but realistic store content.

Try it

- Install: pypi.org/project/moorcheh-edge
- Docs: docs.moorcheh.ai/on-edge
- Demo repo: github.com/moorcheh-ai/retail-kiosk
- UNO Q guide: docs.moorcheh.ai/on-edge/guides/arduino-uno-q

---

13. Appendix: glossary

Term	Meaning
RAG	Retrieval-augmented generation
MIB	Maximum Information Binarization - vector compression to compact one-bit codes at upload
EDM	Efficient Distance Metric - similarity on quantized codes at query time
kiosk_mode	Stricter score filter for voice UX
BGE	BAAI/bge-small-en-v1.5 embedding model (384-d)
