ok, next, explain ollama to me. i have a basic understanding, so maybe go into the interface between ollama, the model(s), and the api it exposes to the outside

Got it — let's go deeper into **Ollama** and how it sits between models and the outside world.

---

1. What Ollama Is

Ollama is essentially a **local model runner and orchestrator**.

- It lets you run large language models (LLMs) on your own machine, with optimizations for Apple Silicon (M1/M2/M3) but also working on Linux/Windows with CUDA.
- It's designed to make models **portable and callable** through a lightweight interface, without needing to build your own PyTorch or TensorRT pipelines.
- Think of it as a "runtime + API wrapper" for open-source models.

---

2. The Core Architecture

At its heart, Ollama sits between three layers:

a) Model Layer

- Ollama packages models into `.ollama` files (bundles) that include the **model weights, config, and prompts**.
- Models can be based on LLaMA, Mistral, Phi, Gemma, etc.
- Ollama manages **quantization** (like 4-bit, 8-bit) so models fit on consumer GPUs/CPUs.

b) Runtime/Interface Layer

- Ollama runs a **local service** (by default on port `11434`) that exposes a **REST API**.
- This service loads the model into memory, manages GPU/CPU allocation, and handles token streaming.

- It also supports **model switching** on demand — you can load different models into memory without rewriting your code.

c) Client/API Layer

- External tools (your app, curl, Python, Node.js, etc.) talk to Ollama via **HTTP calls**.
- Example:

```
curl http://localhost:11434/api/generate -d '{ "model": "llama2",
"prompt": "Why is the sky blue?" }'
```

- Response is **streamed JSON**, so you can start consuming tokens as they are generated.
- This design makes Ollama act like a **drop-in API** similar to OpenAI's — except the models run entirely on your own hardware.

---

3. How Data Flows

Here's a simplified picture:

```
Your App (Python/JS/CLI)
        |
   REST API (localhost:11434)
        |
   Ollama Runtime (model manager, tokenizer, quantization, GPU bindings)
        |
   Model Weights (GGUF / safetensors)
        |
   Hardware (GPU/CPU cores, CUDA/Metal backends)
```

So the flow is:

1. Your app sends a prompt over HTTP.
2. Ollama runtime handles tokenization, batching, GPU calls.
3. Model generates tokens, streamed back over HTTP as JSON chunks.
4. Your app consumes the stream in real time.

---

4. Why This Matters

- **Developer Simplicity** → you don't have to write CUDA kernels or worry about tensor formats.
- **Portability** → same API whether you run a 7B model on a laptop or a 70B on a workstation.
- **Privacy** → everything is local, no cloud inference required.
- **Compatibility** → easy to swap in different open-source models without changing your code.

⚡ So in short: **Ollama = local model server + API wrapper**. It abstracts away the hard parts of model loading and GPU interfacing, exposing a clean API that looks almost identical to cloud services.

Would it help if I drew you a **mermaid diagram** (like the SAGE one) that maps out Ollama's role between your app, the runtime, the models, and the GPU? That might make the data flow even clearer.

tokenization - what is the token format between ollama and the model? how and where is it defined? in both directions.

Great question — you're digging into the "guts" of how Ollama actually hands text back and forth to the model. Let's break it down carefully.

---

1. What "Tokenization" Means Here

- Models don't operate on raw text strings — they work on **tokens** (integers representing pieces of text).

- A tokenizer maps between **human-readable text** and the **integer IDs** the model expects.
- So between Ollama and the model, the format is always **integer token IDs** → processed through embeddings → run through the transformer → output IDs → detokenized to text.

---

2. Where the Token Format Comes From

Each model architecture comes with a **specific tokenizer definition**, which is bundled with the model weights.

- **LLaMA-family models** (Meta's LLaMA, Mistral, etc.) use **SentencePiece** tokenizers. These are stored in a `.model` file (or embedded into the GGUF/GGML format used by Ollama).
- **GPT-like models** (Phi, Gemma, Falcon, etc.) may use **Byte Pair Encoding (BPE)** or a variant, also stored in model metadata.
- Ollama doesn't invent its own tokenizer — it **uses the one defined by the model itself**. That ensures integer IDs map correctly to what the transformer expects.

---

3. Inside Ollama

- When you send a request (say `"prompt": "Hello world"`), Ollama:
  1. Looks up the model's tokenizer (from the model package).
  2. Runs text → integer token IDs.
  3. Sends those IDs into the model's inference loop.
- On the way back out:
  1. Model generates integer token IDs.
  2. Ollama runs detokenization (IDs → text).
  3. Streams the text back to your client as JSON chunks.

So, **between Ollama and the model: it's integers. Between Ollama and you: it's text wrapped in JSON.**

---

4. Example (LLaMA-style model)

Let’s say the prompt is `"Why?"`

1. Tokenizer might split it into:
   - `"▁Why"` → ID `3102`
   - `"?"` → ID `29`

   So Ollama actually sends `[3102, 29]` into the transformer.
2. The model generates `[3102, 364, 50256]` → which detokenizes to `"Why not"` (then `50256` is an EOS token).
3. Ollama streams back:

```
{"response": "Why not", "done": true}
```

---

5. Where It’s Defined in Ollama

- The **tokenizer files** are baked into the `.ollama` model bundle (usually GGUF format).
- Ollama’s runtime loads the tokenizer directly from the model metadata.
- Implementation detail: Ollama leans on `llama.cpp` under the hood, which has support for GGUF/GGML models and their tokenizers.

So: the **tokenization scheme is defined by the model’s architecture**, and Ollama simply enforces that scheme when passing text → model and model → text.

✅ **Summary:**

- Between Ollama and *your app*: UTF-8 text (via JSON).
- Between Ollama and *the model*: integer token IDs.
- The mapping is defined by the model’s tokenizer (SentencePiece/BPE), included in the model bundle (GGUF/ollama package).

Would you like me to make you a **diagram** (like the others) that explicitly shows the two directions — text ↔ token IDs ↔ transformer ↔ token IDs ↔ text — so you can drop it straight into your repo as documentation?

so if we want sage to talk to a model, it would have to get its tokenizer info and we'd need a 'dictionary entity' essentially which maps between text (with to/from speech layers if applicable) and a specific model's token ids. or between different model token ids (does that have to go through text?)

Exactly — you’ve put your finger on the right abstraction layer. Let me unpack it in terms of **SAGE ↔ Model communication**:

---

1. Tokenization as the Bridge

Every model has its own tokenizer → a mapping of **text ↔ token IDs**.

- For SAGE to talk to a model, it must **import the tokenizer definition** (SentencePiece, BPE, etc.) that comes with that model.
- Without that, you’d send the wrong integers and the model would output nonsense.

So yes, you’d need a **“dictionary entity”** in SAGE, tied to each model, that knows how to:

- Encode (text → tokens)
- Decode (tokens → text)

---

2. Speech Layers

If speech is involved (say audio input/output):

- You’d first run **speech-to-text (ASR)** to produce text, then tokenize into IDs for the target model.
- On the way out, you’d detokenize IDs → text → **text-to-speech (TTS)**.

That way, speech layers just wrap the text layer, and the model itself always sees token IDs.

---

3. Between Different Models

If you want **Model A** (with tokenizer A) to talk to **Model B** (tokenizer B), there are two approaches:

a) Through Text (safe, universal)

1. Detokenize A's output tokens → text.
2. Retokenize that text with tokenizer B → IDs.

- This is the **robust option** (universal compatibility).
- Downside: introduces a detokenization → re-tokenization cost.

b) Direct Token Mapping (fast but tricky)

- You could build a **cross-token dictionary** if both tokenizers are based on similar subword units.
- Example: If both use SentencePiece with overlapping vocab, you could map tokens directly without detokenizing.
- This is fragile: any mismatch in how vocabularies split words → meaning drift.
- Useful for performance-critical pipelines, but you'd need alignment tables.

---

4. What SAGE Needs

So, in the SAGE framework you'd define something like:

```
class DictionaryEntity: def __init__(self, tokenizer): self.tokenizer =
tokenizer def encode(self, text: str) -> List[int]: return
self.tokenizer.encode(text) def decode(self, tokens: List[int]) -> str: return
self.tokenizer.decode(tokens) def translate(self, tokens: List[int],
target_dict) -> List[int]: text = self.decode(tokens) return
target_dict.encode(text)
```

- Each model gets its own `DictionaryEntity`.
- If SAGE wants to mediate between models, it either goes through text or (later optimization) through precomputed token maps.

✅ Bottom line:

- Yes, SAGE needs **dictionary entities** to bridge between text, speech, and token IDs.
- Safest path between different models is **tokens → text → tokens**.

- Advanced optimization might do **direct token maps** between vocabularies, but that's a second step.

Would you like me to write up a **design doc snippet in Markdown** (with diagram) that you could drop straight into your SAGE repo as "Tokenizer & Dictionary Abstraction"? It would frame this as an architectural component alongside HRM, Sidecar, and GPU Mailboxes.

not yet. in web4, a dictionary is an active entity with its trusted domain, provenance, context, etc. when we view sage as a fractal self-contained web4 instance (interfacing with outside via sensors/effectors), within its own ecosystem we can have many dictionaries. a speech to text interface is a form of a dictionary. it would be trusted to translate between audio module and text (let's keep it english for now). it would not be trusted for much else. the memory sensor would need to remember that, so that when sage gets input from audio sensor, it (through prior training) decides this is speech, do i recall how to deal with that? if yes, is the trusted dictionary loaded (in gpu or cpu, a dictionary would have preferred device as part of its metadata). if yes, ask it to confirm that it trusts itself to interpret what it hears. if yes, the pipeline gains operational trust threshold and is monitored from there. if any of those are no, sage either attempts to build the pipeline (by finding and loading the trusted dictionary), or just says this audio is low trust, i don't know what to do with it in detail, but it may be useful peripherally (direction, intensity, patterns, etc).

Love this framing. Treating **dictionaries as active, trusted translators** inside a fractal Web4/SAGE instance clicks perfectly with what we've built (mailboxes, trust maps, memory sensor). Here's a tight design you can drop into the architecture without creating new tech debt.

Dictionary Entities inside SAGE (Web4-style)

1) What a Dictionary Is (in SAGE)

An **active module** that translates between two modalities or codebooks, with its own:

- **Trusted domain** (what it's allowed to translate: audio→english-text, tokensA→tokensB, pixels→symbols, etc.)
- **Provenance** (who/what produced it, version, license, build hash)
- **Context** (operational assumptions: language, sampling rate, tokenizer version)
- **Preferred device & cost** (GPU/CPU/NPU, memory footprint)
- **Self-assessment hooks** (it can say "I'm confident" or "I'm out of domain")

Think of tokenizers, ASR, TTS, OCR, VAD, phoneme mappers, and even **model-to-model token bridges** as different Dictionary types.

2) Minimal Interface Contract

```
class Dictionary: id: str # stable UID version: str domain_in: str # e.g.,
"audio/pcm@16k", "text/en", "tokens/llama3" domain_out: str # e.g., "text/en",
"tokens/mistral" provenance: dict # repo, commit, signer, attestation context:
dict # language, tokenizer hash, sample rate... preferred_device: str #
"gpu"/"cpu" footprint_mb: int trust_cap: float # max trust this dictionary can
contribute (0..1) def can_handle(self, meta: dict) -> bool: ... def
self_check(self) -> dict: ... # {ok:bool, conf:0..1, reason:str} def
translate(self, x, meta: dict) -> (y, meta_out, conf: float)
```

Operational **trust** at runtime is *not* a property of the dictionary alone; it's computed from:

- prior **experience** (memory sensor),
- **self_check** confidence,
- **context match** (sample rate/lang/tokenizer hash),
- recent **outcome signals** (SNARC: Surprise/Novelty/Arousal/Reward/Conflict).

3) Placement in the SAGE loop

```
[Physical Sensors]  ->  L-module sweep  ->  H-module strategy
     |                        |                   |
 [Audio] ---+           [Dictionary Registry]     |
            |                 |                   |
            +-> ASR dict? -> text -> tokenizer -> cognition
                     \-> no? mark low-trust peripheral features (VAD, energy)
```

- **Registry**: index of known dictionaries (by domain_in/out, context, device).

- **Memory sensor**: stores episodes like "audio@16k → ASR/en conf=0.82 → helped answer Q".
- **Trust map**: learned weighting that gates whether to invoke a dictionary and how much to rely on its output.

4) Runtime Flow (speech example)

1. **Detect**: audio sensor posts mailbox event with meta `{sr:16k, lang:unknown}` .
2. **Recall**: H-module queries memory: "for audio@16k, which pipelines worked?"
3. **Select**: Registry returns candidate dicts: `VAD` , `ASR(en)` , `ASR(multilang)` .
4. **Gate by trust**:
   - if memory says `ASR(en)` had high utility in similar context → try **self_check()**.
   - if self_check ok & device available → **translate()** (audio→text), attach `conf` .
5. **Monitor**: downstream reasoning validates text (coherence with other sensors, user feedback, task success). Emit SNARC signals to memory.
6. **Adjust**: trust map updates association: `(context, dict_id) ⟶ utility` .

If any step falls below threshold:

- Demote to **peripheral** processing (keep energy patterns/direction cues),
- Or try an alternate dict (e.g., `ASR(multilang)`).

5) "Tokenizer as Dictionary"

Exactly the same pattern:

- `Tokenizer(english_text → tokens/llama3)`
- `Detokenizer(tokens/llama3 → english_text)`
- **Model-to-model bridge** dictionaries:
  - Safe path: `tokensA → text → tokensB`
  - Fast path (optional): **direct token map** with coverage/ambiguity metrics; expose `self_check()` that drops confidence when encountering OOV or misaligned merges.

This lets SAGE run **multiple LLMs** as cognitive sensors while keeping pipelines explicit and trust-mediated.

6) Data/Meta on the Mailboxes

Peripheral broadcast (cheap signals):

```
{kind:"audio_vad", sr:16000, energy:0.41, direction:~75deg, trust_hint:0.3}
```

Focus mailbox (rich tensors/payload pointers):

```
{kind:"asr_text", ptr:dev_ptr, bytes:N, lang:"en", conf:0.86, dict_id:"asr_en@v1.
 tokenizer_hash:"abc123", timecode:[t0,t1]}
```

Downstream components don't "assume" text is correct—they read **conf** and **dict_id** and let trust decide.

7) Trust Computation (sketch)

```
trust(dict, context) =
   clamp( w0 * prior_memory_score(dict, context)
        + w1 * dict.self_check_conf
        + w2 * context_match(dict.context, context)
        + w3 * recent_reward_signals
        - w4 * conflict_signals, 0, dict.trust_cap )
```

- **prior_memory_score** learned over episodes; decays with time / context distance
- **dict.trust_cap** prevents any single dictionary from dominating early
- **conflict signals** (e.g., text contradicts vision) reduce trust fast

8) Sleep/Dream hooks

- During sleep, synthesize **counterfactual episodes**: corrupt audio, accent shifts, noise profiles; measure how the **ASR dictionary** conf and downstream utility change → update trust curves.
- For token bridges, generate paraphrases / edge cases; refine coverage metrics.

9) Failure Modes & Mitigations

- **Out-of-domain input**: `self_check()` flags low conf → pipeline either routes to alternate dict or degrades gracefully to peripheral cues.
- **Resource starvation**: registry knows **preferred_device**; scheduler can demote GPU-heavy dicts if focus budget tight.
- **Version drift** (e.g., tokenizer change): provenance + hash mismatch → trust penalty or block until retrained.

10) Tiny state machine (per input burst)

```
IDLE
 └─▶ DISCOVER (look up candidate dicts by domain_in)
        ├─▶ RECALL (query memory for prior success)
        ├─▶ VERIFY (dict.self_check; context match)
        ├─▶ DECIDE (trust ≥ τ?) ─▶ APPLY → MONITOR → LEARN → IDLE
        └─▶ else ─▶ PERIPHERAL (store low-level features) → IDLE
```

11) Minimal control pseudocode

```
def route_input(ev): cands = registry.find(ev.domain_in) ranked =
rank_by_memory_and_context(cands, ev.meta) for d in ranked: if not
d.can_handle(ev.meta): continue chk = d.self_check() t = compute_trust(d,
ev.meta, chk.conf) if t < THRESHOLD: continue y, meta_out, conf =
d.translate(ev.payload, ev.meta) post_focus(kind=meta_out['domain'],
payload=y, meta={**meta_out, 'dict_id':d.id, 'conf':conf})
monitor_and_update_trust(d, ev, y, conf) return
post_peripheral(kind='low_trust_'+ev.domain_in, features=cheap_features(ev))
```

Why this matches your Web4 framing

- **Dictionaries are first-class citizens** with trust, provenance, and domain constraints.
- **Fractal**: SAGE is a self-contained Web4 instance—sensors/effectors + active translators + trust/strategy.
- **Memory makes it adaptive**: it's not "always use ASR"—it's "use ASR in contexts where it has earned trust."
- **GPU-native**: preferred device + mailbox payloads keep hot paths on-device.

If/when you want, I can turn this into a **short design doc + diagram** (MD) that you can drop into the repo next to the mailbox docs.

excellent, though we need to make the registry even more generic as 'entity registry' where dictionaries are just one type of entity. we also need an entity memory, which is a registry of available (installed or not) entities, a sidecar style retrieval mechanism, and a reputation history. fundamentally, an entity is a sensor, an effector, or both. dictionary is a subtype of entity. it is a sensor for its input and effector for its output, so it's a 'both' entity. 'dictionary' is a role the entity plays in the ecosystem. write a detailed downloadable md that starts with overall sage architecture framed as a web4 instance, drill down to entities, roles, and use dictionaries (several types) as concrete examples.

Here's the file you asked for, with the **expanded architecture doc** showing SAGE as a Web4 instance, drilling down to entities and dictionary roles:

📁 Download SAGE_ENTITIES_AND_DICTIONARIES.md

This version:

- Frames **SAGE as a fractal Web4 instance**.
- Defines **entities** as the universal building blocks (sensor, effector, or both).
- Introduces **Entity Registry** (runtime) and **Entity Memory** (persistent reputation/history).
- Uses **dictionaries** as concrete examples of "both" entities with roles (ASR, tokenizer, cross-model bridge, TTS).
- Shows how **trust and reputation** flow into decision-making.
- Includes a **Mermaid diagram** of the workflow.

Would you like me to also draft a **companion document** called something like `ENTITY_MEMORY.md`, spelling out how provenance, trust episodes, and retrieval hooks would be stored/queried? That would give Claude a concrete spec to start stubbing out in code. </>

yes, please

Here you go — I wrote up the full spec as requested:

- 📄 **ENTITY_MEMORY.md** — Download

It covers:

- Purpose and framing within SAGE as a Web4 instance
- Data model (entity records, episodes, retrieval records) with JSON-like schemas
- Trust/reputation math (context keying, decay, SNARC-weighted scores)
- APIs (CRUD, queries, runtime helpers), storage backends, and indexing
- Mailbox integration (peripheral and focus examples)
- Sleep/dream consolidation workflow
- Security/integrity considerations
- Performance guidance (Jetson-first)
- Mermaid diagrams for runtime and sleep flows
- Pseudocode stubs that Claude can turn into real code

Want me to follow up with a **SQLite schema + minimal Python implementation** (so you can run it on Jetson right away), or leave it at the spec for now? </>

let's leave it as spec, i want to make sure it's consistent with sidecar (to the extent necessary/practical), and a few other things. this is a great start, thank you!

Glad this hits the mark 🙏

You're absolutely right to hold it at **spec level** for now — that way we can check alignment with:

- **Sidecar memory mechanisms** (selective retention, SNARC-gated writes, associative recall)
- **Web4 trust spectrum** (entity reputation needs to map cleanly)
- **GPU mailbox architecture** (where runtime trust scores may act as routing weights)

The spec already has placeholders where those can be slotted in, but consistency across systems is more important than rushing into code. Once you decide how much overlap vs. separation you want between **Sidecar** and **Entity Memory**, we can refine:

- whether Entity Memory is a **superset** (holding sidecar traces, entity registry, and provenance)
- or a **parallel module** (with its own database but cross-indexes into sidecar episodes)

That distinction will drive the first schema choices.

Do you want me to sketch a **comparison table** (Entity Memory vs. Sidecar) to make it clearer where they overlap, where they differ, and where integration points are? That might help you see whether to merge them, keep them modular, or link them with a "bridge entity."

yes, that would be helpful

Here's a clear side-by-side to help decide how Entity Memory should relate to Sidecar.

Entity Memory vs. Sidecar — Comparison & Integration Map

Dimension	Entity Memory	Sidecar (Transformer-Sidecar)	Notes / Implicat
Primary purpose	Registry + reputation for *entities* (sensors/effectors/dictionaries); provenance; availability; trust over contexts	Selective, affect-gated *experience memory* for the agent; fast associative recall	Separate concern *to use*" vs "*what* *experienced*"
Core objects	Entities, roles, episodes (entity use), reputations, retrieval plans	Traces, keys, embeddings, eligibility/affect signals (SNARC)	EM uses Sidecar episodes as evid signals
Granularity	Coarser: per entity/pipeline, per context bucket	Finer: per span/chunk/interaction, token-level or segment-level	EM aggregates; stores raw-ish
Temporal focus	Long-horizon trends & decay (days→months)	Short→mid horizon recency &	Complementary constants

Dimension	Entity Memory	Sidecar (Transformer-Sidecar)	Notes / Implicat
		consolidation (seconds→days; sleep compaction)	
Trust math	Reputation curves, context-conditioned priors, caps; conflict/staleness penalties	SNARC-weighted salience → write/keep; recall strength	EM's trust can cc Sidecar's SNARC signals
Read path	Query by role/domain/context to *select entities* and weight them	Recall by key/situation to *retrieve experiences*	Selection (EM) th reasoning (Sidec aided)
Write path	Append "entity episode" with outcome & SNARC; update stats/decay	Append traces opportunistically; consolidate during sleep	Both are append different schema
Provenance	Strong: signer, build hash, license, device prefs, install state	Weaker/implicit: context of traces, model snapshots	EM is the proven ground truth
Availability	Knows installed vs retrievable; sidecar-style *retrieval plan* for entities	N/A (except where memories persist on disk)	EM can request installs/loads
Device placement	Host DB (SQLite/LMDB/DuckDB); hot indices cached; GPU-side hints mirrored	GPU-adjacent vector stores; on-device KV for fast recall	Split placement; ties them
APIs	Search/select entities; record episode; get reputation; get install plan	Put/recall trace; consolidate; similarity search	Clean seams; kee small
Mailbox use	Heartbeats, entity_missing, episode_result → update reputation	Recall hits, salience events, sleep summaries → influence EM	Both publish/sub decouple modul

Dimension	Entity Memory	Sidecar (Transformer-Sidecar)	Notes / Implicat
Sleep role	Batch update reps (low weight for synthetic), decay, snapshot	Generate augmentations, replay, compress, link traces	Joint nightly cycl
Security	Attestation & policy caps; quarantine on anomaly	Data hygiene; PII controls in traces	EM enforces poli Sidecar respects
Failure modes	Bad priors/overtrust; stale reputations	Over-retention/noisy recall; drift	Cross-checks: co reduces trust; sle prunes

---

Integration Patterns (pick one)

A) Parallel Modules + Bridge (recommended)

- Keep **Entity Memory (EM)** and **Sidecar** separate.
- Add a thin **Bridge** that:
  - consumes Sidecar SNARC summaries as *signals* for EM's reputation updates,
  - publishes EM trust priors back to the scheduler/selector.
- Pros: clean separation, easy to evolve each; minimal coupling.
- Cons: one more tiny component.

B) Entity Memory as Superset

- EM owns entities **and** stores a *summary view* of Sidecar traces (not raw).
- Sidecar still holds raw traces; EM periodically ingests aggregates.
- Pros: fewer moving parts at API surface.
- Cons: schema bloat; risk of mixing concerns.

C) Sidecar-backed Feature Store

- Sidecar exposes a "reputation features" view; EM is a thin index over it.
- Pros: maximum reuse of Sidecar infra.
- Cons: EM becomes dependent on Sidecar uptime/format.

My vote: A) Parallel + Bridge — clearest contracts, lowest coupling.

Minimal Bridge Contracts

From Sidecar → EM (signals):


```
{ "kind":"snarc_summary", "episode_id":"ep:...",
"entity_id":"entity:asr/openen@1.3#...", "ctx_key":"audio@16k|lang=en",
"score_components":{"reward":0.9,"coherence":0.78,"conflict":0.1},
"latency_ms":95, "ts":"..." }
```


From EM → Scheduler/H-module (priors):


```
{ "kind":"entity_prior", "entity_id":"entity:asr/openen@1.3#...",
"ctx_key":"audio@16k|lang=en", "trust_prior":0.86, "trust_cap":0.92,
"staleness_penalty":0.03 }
```

Decision Heuristics (when to consult which)

- **Choosing *who* to invoke** (ASR vs. multilingual ASR vs. “low-trust peripheral”): **EM**
- **Choosing *what* to recall** (similar past utterance/context): **Sidecar**
- **Nightly updates**: Sidecar synthesizes; EM adjusts reputations with low weight
- **Conflicts** (vision contradicts ASR): EM applies conflict penalty; Sidecar retains the episode with *conflict tag* for future analysis

---

Example Pipeline (speech → cognition → speech)

1. **Registry** lists available ASR entities (installed + retrievable).
2. **EM** ranks by context (16k/en, device budget).
3. Selected ASR runs → **Sidecar** logs traces & SNARC.
4. LLM reasoning uses Sidecar recalls (similar past Q&A).
5. **EM** records entity episode with outcome (reward/coherence/latency).
6. **TTS** selection repeats steps 1–5 for output path.
7. Sleep: Sidecar augments/replays; EM updates reputations.

If you want, I can turn this into a **downloadable MD** with the table and contracts (to drop straight into the repo), or fold it into the `ENTITY_MEMORY.md` as an appendix.