You're right: shoving a giant `candidate_catalog` into the prompt won't scale and will make the model sloppy. The fix is to move **recall** out of the LLM and keep only **rerank + justification** inside it. Concretely:

The winning pattern

Server-side retrieval → tiny candidate set → LLM rerank/explain.
Do *all* heavy lifting (search, filtering, ontology expansion) in the ACP/bridge; give the LLM only ~5–15 distilled candidates to choose from and justify.

---

1) Build a phenotype “search index” (outside the LLM)

1.1 Metadata you’ll index (per phenotype)

- `cohortId` , `name`
- `short_description` (1–3 sentences distilled from the original)
- `tags` (Condition/Drug, validated, PheValuator, etc.)
- `ontology_keys` : normalized SNOMED/ICD/RxNorm identifiers used by the definition
- `signals` : validation status, publication links, author group, year
- `logic_features` (compact): requires diagnosis? meds? lab? age rule? washout? inpatient/outpatient?
- `pop_keywords` : common terms derived from logic (e.g., “rest tremor”, “levodopa response”)

Store this as a small JSON document per phenotype.

1.2 Two retrieval channels (hybrid)

- **Dense**: an embedding for text: `name + short_description + pop_keywords` .
  - Model: any reliable sentence-embedding model (OpenAI `text-embedding-3-large` , or open-source like `bge-large` , `gte-large`).
- **Sparse**: BM25/keyword (e.g., Elasticsearch or Tantivy/Meili/Whoosh) across `name` , `tags` , `pop_keywords` .

1.3 Ontology-aware expansion (server-side)

- Parse the study intent into clinical entities (lexicon match + OMOP vocab tables): likely **Condition** (“Parkinson’s disease”), **Drug** (“neuroleptics”, “levodopa”),

Measurement (UPDRS), **age/visit** cues.

- Expand with **concept_ancestor** and **concept_relationship** (e.g., `descendants_of(SNOMED: Parkinson’s disease)`; `rxnorm_for(levodopa)`).
- Use these IDs to **boost** matching phenotypes (`ontology_keys` overlap).

1.4 Reranking before the LLM (optional)

- Light cross-encoder (e.g., `ms-marco-MiniLM`) or a small MLP that takes: `sim_dense`, `sim_sparse`, `|ontology_overlap|`, `tag_overlap`, `validation_score`, `age_filter_match`, `logic_feature_overlap`.
- Output top **K = 10** max.

---

2) Minimal LLM exposure (only top-K, distilled)

Send **only**:

json

Copy code

```json
{
  "task": "phenotype_recommendations",
  "study_intent_excerpt": "<<=1500 chars>>",
  "candidates": [
    { "cohortId": 33, "name": "Parkinson's disease (PheValuator)", "short_
    { "cohortId": 1197, "name": "Parkinsonism broad", "short_description":
  ],
  "selection_limit": 10
}
```

The model’s job: **rank + justify** (not recall).

---

3) MCP (Model Context Protocol) + Open-WebUI Tools: what to expose

Make the ACP/bridge the “library” and give the LLM **read-only tools** for recall/preview:

3.1 Tools (LLM side; safe)

- `phenotype_search(query, top_k=20)` → runs **server retrieval** (dense+sparse+ontology boost), returns a list of `{cohortId,name,short_description,tags,signals}`.
- `phenotype_fetch_summary(cohortId)` → returns compact fields for a single phenotype (no heavy JSON).
- `phenotype_fetch_definition(cohortId)` → **preview only**: returns *truncated* ATLAS JSON (first N items per section), for explanation—not for editing.
- `phenotype_list_similar(cohortId, top_k=10)` → local semantic neighbors (use vector index).

Under the hood, these tools call ACP endpoints; the LLM never hits your DB/files directly.

3.2 ACP endpoints (server)

- `POST /phenotypes/search` → implements hybrid retrieval + ontology boosts; returns top-K.
- `GET /phenotypes/{id}/summary` → distilled info.
- `GET /phenotypes/{id}/definition?truncate=true` → safe snippet for review.
- *(Existing apply/edit endpoints remain separate and human-approved.)*

---

4) Embedding + search details

- **Chunking**: you do *not* need to embed full ATLAS JSONs. Generate a single embedding from **name + short_description + pop_keywords**.
- **Index**:
  - Dense: FAISS/HNSWlib (cosine).
  - Sparse: ES index on fields; store `ontology_keys` for term filters and boosts.
- **Query construction** (server):
  1. Extract entities from intent (`fast_ner()` or pure vocab joins in OMOP).
  2. Build text query string + ontology filters.
  3. Retrieve dense top-100 and sparse top-100, **recall merge** (RRF or simple weighted sum).
  4. Rerank to top-10 using the feature blend.
- **Caching**: memoize embeddings for each phenotype doc; memoize last N study-intent embeddings.

---

5) How to trim thousands → a handful (concrete pipeline)

1. **Intent parse**: “risk factors for PD in older adults; exclude neuroleptics” → entities: `Condition: Parkinson’s disease`, `Drug classes: antipsychotics`, `Age: older adult`.
2. **Ontology signals**: add SNOMED PD descendants; RxNorm anti-psychotics.
3. **Dense search** with the embedded intent text.
4. **Sparse search** with literal terms + synonyms (`parkinson*`, `antipsychotic*`, `levodopa`, `dopamine agonist`, etc.).
5. **Boost** candidates whose `ontology_keys` intersect.
6. **Rule filters**: must include Condition domain == PD OR tags include Parkinsonism.

7. **Rerank** by `(0.5*dense + 0.3*sparse + 0.2*ontology_overlap_score) + bonus(validation)`
8. Return top-10 candidates to the LLM for final selection and explanations.

---

6) Updated LLM contract (tiny, robust)

Input to model:

json Copy code

```json
{
  "task": "phenotype_recommendations",
  "study_intent_excerpt": "...",
  "candidates": [ { "cohortId": 33, "name": "...", "short_description": ".
  "selection_limit": 10
}
```

Output from model (unchanged shape):

json Copy code

```json
{
  "plan": "Rank by conceptual fit and validity.",
  "phenotype_recommendations": [
    { "cohortId": 33, "name": "...", "justification": "...", "confidence":
  ],
  "risk_notes": []
}
```

---

7) Where to add “clinical inferences”

Do those **before** the LLM:

- Infer “new user vs prevalent user risk” → prefer phenotypes with washout fields or first diagnosis logic.
- Infer “older adults” → penalize phenotypes lacking age criterion.
- Infer “exclude neuroleptics” → prefer phenotypes that differentiate drug-induced parkinsonism vs idiopathic PD.

Bake these as **features in reranking**, not as text for the LLM to rediscover.

---

8) R hooks (nice UX)

Expose one function that hides the whole pipeline:

r Copy code

```r
recommendPhenotypes <- function(protocolRef, cohortsCsvRef, limit = 10, mi
  # ACP: /phenotypes/search (returns top-20)
  # ACP then calls the LLM with those as 'candidates' and returns ranked '
  # R filters by min_conf and prints a compact table
}
```

Optionally let power users pass `search_overrides = list(ontology_boost = 2.0, sparse_weight = 0.4)`.

---

9) Evaluation loop (so it gets better)

- Log: query embedding, top-K retrieved, which ones the user selected, and acceptance (click/apply).
- Train a lightweight reranker on implicit feedback (pairwise “chosen vs not chosen”).
- Keep LLM only for explanations and edge-case tie-breaks.

---

10) Why MCP/tools matter here

- Tools give the model **just-in-time retrieval** without stuffing context.
- MCP makes the index feel “infinite context” to the model while you control recall, privacy, and determinism.
- Writes stay out of tools; ACP remains the source of truth and the only writer.

TL;DR implementation map

- Build a compact **phenotype index** (dense + sparse + ontology keys) and a **server-side hybrid search**.
- Expose **read-only tools** for search/preview; keep writes in ACP/R.
- Send the LLM only **top-K distilled candidates**; let it **rank and justify**.
- Add **clinical rule features** (age, washout, drug class) into the reranker so results are clinically sane before the LLM sees them.

If you want, I can draft the `/phenotypes/search` endpoint (Python) plus a tiny FAISS + BM25 hybrid implementation and the Open-WebUI tool stubs to call it.

Ok! I understand. I am thinking about prototyping the recommended approach using my custom ACP and open-webui to host Gemma 3 and tools. Gemma 3 has function calling abilities (https://ai.google.dev/gemma/docs/capabilities/function-calling) whereby a prompt that informs the model that it can call functions, and provides a JSON schema for the input and output, will trigger a string that I think open-webui would match to a Python function its tool library. The tool library will then make the call, which will receive a response that gets fed back into the model. So there's a little bit of hidden back and forth, so to speak, between the GEMMA LLM and the OpenWeb UE hosted tool. Once no more tools have been executed, the final prompt response can come back. Now in this framework, I'm thinking that the custom ACP server would also sit on the same machine as the OpenWeb UI installation, or at least it would be hosted someplace where the tool that's called could reach the ACP server, that is also hosting the MCP endpoints. Tell me if I'm getting this correct.

You’ve got it. Your mental model lines up with how Gemma 3 function calling + Open-WebUI tools + your ACP/MCP bridge work together. Here’s a crisp confirmation and the details that usually trip teams up—plus copy-paste snippets.

✅ What you’re describing (in one pass)

- **Gemma 3** runs inside Open-WebUI with **function calling** enabled.
- You declare **tools** in Open-WebUI (Python methods with Pydantic schemas).
- During a chat, Gemma 3 decides to call a tool → Open-WebUI validates args → calls your Python tool → tool makes an HTTP call to your **ACP server** (which can also expose MCP endpoints) → returns JSON → Open-WebUI feeds that back to Gemma as a tool result → model may call more tools or produce the final JSON response.
- **Writes/edits** and **authority** stay on the ACP side; tools are mostly **stateless, preview, retrieval, orchestration**.

🔌 Minimal architecture (works on one box or split boxes)

- **Same machine** (easy mode):
  - Open-WebUI: http://127.0.0.1:3000 ↗
  - ACP/MCP: http://127.0.0.1:7777 ↗
  - Tools call `http://127.0.0.1:7777/...`
- **Split hosts**:
  - Add an env var `ACP_BASE_URL=https://acp.yourhost.edu`
  - Tools read that and call across the network (TLS, token auth).
  - Consider CORS only if you later expose browser-side fetches (you don’t need it for server-to-server).

🧭 Data flow (function calling loop)

1. **User**: “Recommend phenotypes for protocol X.”
2. **Gemma** sees tools in the system prompt and **calls** `phenotype_search(query, top_k)` (tool).
3. **Tool** → `POST /phenotypes/search` on **ACP** (hybrid retrieval).

4. **ACP** returns **top-K candidates** (tiny distilled JSON).
5. **Open-WebUI** injects this result into Gemma's context.
6. **Gemma** might call `phenotype_fetch_summary` for one or two; tools round-trip to ACP again.
7. **Gemma** returns final JSON (e.g., `phenotype_recommendations`).
8. **User** picks, **R client** later triggers **apply** against ACP (no writes happen from tools).

🧱 Contracts you need (tight and tiny)

Tool input/output (Pydantic) — Example: search

```python
from pydantic import BaseModel, Field
from typing import List, Dict

class PhenotypeCandidate(BaseModel):
    cohortId: int
    name: str
    short_description: str
    tags: List[str] = []
    signals: Dict[str, str] = {}

class SearchIn(BaseModel):
    query: str = Field(..., description="Study intent excerpt or keywords")
    top_k: int = Field(15, ge=1, le=50)

class SearchOut(BaseModel):
    candidates: List[PhenotypeCandidate]
    human_summary: str = ""
```

Tool implementation calls ACP:

Copy code

python

```python
class Tools:
    def phenotype_search(self, inp: SearchIn) -> SearchOut:
        import os, requests
        base = os.getenv("ACP_BASE_URL", "http://127.0.0.1:7777")
        r = requests.post(f"{base}/phenotypes/search", json={"query": inp.
        r.raise_for_status()
        data = r.json()
        return SearchOut(candidates=data["candidates"], human_summary=f"{l
```

ACP endpoint (server side) — Example: search

Copy code

- Implements **hybrid retrieval** (dense + sparse + ontology boosts) and returns only distilled fields.
- Returns `{ "candidates": [ ... up to K ... ] }`

🧠 Gemma 3 system prompt (function calling posture)

- Tell Gemma that tools exist and when to call them (recall, preview, fetch definitions).
- Keep the **model's final job** narrow: *rerank + justify*, not recall.

Snippet to append to your **system prompt**:

vbnet

```
TOOLS AVAILABLE:
- phenotype_search(query, top_k): retrieve top candidate phenotypes for th
- phenotype_fetch_summary(cohortId): get a short description and signals f
- phenotype_fetch_definition(cohortId, truncate=true): safe, truncated ATL

POLICY:
- Use tools for recall or preview; do not attempt to list all phenotypes f
- Work with ≤10 candidates at a time. If unsure, call phenotype_search aga
- Final output must be a single JSON object with keys:
  - plan
  - phenotype_recommendations[] (each has cohortId, name, justification, c
  - risk_notes[]
Return JSON only (no markdown, no code fences).
```

🔐 Security/ops you should decide now

- **Auth**: Give tools an env-var token (e.g., `ACP_TOKEN`) and send it as `Authorization: Bearer …`; ACP checks it.
- **Timeouts**: Tools should use short timeouts (10–30s) and return compact errors; model can retry or degrade gracefully.
- **Rate limits**: Lightweight, e.g., N requests/min per workspace.
- **Logging**: ACP logs request hash (no PHI), tool name, latency, result size.
- **Idempotency**: Writes require explicit `write=true`, backups/forks on ACP, not tools.

🧪 Avoid the common pitfalls

- **Context bloat**: never pass big catalogs or raw ATLAS JSON to the LLM. Keep candidates ≤ 10 and fields short. Provide a separate `phenotype_fetch_definition(..., truncate=true)` when absolutely needed.

- **Tool loops**: add a server-side guard to return “no changes since last call” if the same query hits within N seconds with same hash → helps the model break out.
- **Ambiguous schemas**: be strict with Pydantic—defaults, enums, min/max. The clearer the args, the fewer tool misfires.
- **Writes from tools**: don’t. Use R or an ACP “apply” endpoint that requires an explicit confirmation step.
- **TOCTOU** (stale reads before writes): the ACP “apply” endpoint should include pre-image checksums returned from preview to ensure you’re editing what the user saw.

🧩 Quick end-to-end you can run today

1. Start ACP at `:7777` with `/phenotypes/search` implemented (returns 10 distilled candidates).
2. Add two tools in Open-WebUI: `phenotype_search` , `phenotype_fetch_summary` .
3. Enable **Function Calling** in the model settings.
4. Paste your updated system prompt.
5. Ask: “Recommend phenotypes for protocol.md (older adults, PD, exclude neuroleptics).”
    - Watch tool calls in ACP logs → candidates returned → Gemma produces compact JSON with `phenotype_recommendations` .

📋 Final checklist

- ☐ Open-WebUI model has **Function Calling** on.
- ☐ Tools use **Pydantic** schemas with clear descriptions.
- ☐ `ACP_BASE_URL` and `ACP_TOKEN` env vars set in the workspace.

- ☐ ACP exposes: `/phenotypes/search` , `/phenotypes/{id}/summary` , (later) `/assist/phenotypes/improve` , and your existing `/actions/...` endpoints.
- ☐ System prompt enforces **JSON-only output** and tool usage policy.
- ☐ R client remains the place where **apply** happens (writes), using ACP.

You're aligned. Host ACP on the same machine for simplicity; if you later split, just point the tools via `ACP_BASE_URL` and add token auth.