This is a copy of a conversation between ChatGPT & Anonymous.
Report conversation

I am working on a project that will extend the OHDSI communities tools to provide an agent for developing observational retrospective clinical studies. The principal tools I am focusing on extending are Atlas/WebAPI and R Hades. The approach I am thinking of is to develop an Agent Client Protocol (ACP) server that will enable 1) interaction with an agent via R as the user works with tools such as CohortGeneration, FeatureExtraction, PopulationEstimation, population level estimation, PatientLevelPrediction, etc (see https://ohdsi.github.io/TheBookOfOhdsi/); and 2) interaction within Atlas as a new study-design-by-chat feature. Here is an overview of my rough idea of some of the use cases and features:

Use cases:
- Improve an existing concept set by consideration of the study objectives and by leveraging vocabulary relationships (e.g., in concept_ancestor, concept_relationship), information from the data source(s) that are of interest to the study on concept utilization, and information from the model's parameterized knowledge and other sources about variation and "gotchas" on code usage.
- Import, and possibly adapt/customize, an existing EHR

phenotype definition to better fit the study objectives. The process will result in suggested cohort definition criteria and attributes, awareness of possible performance (PPV, sensitivity, specificity, etc) considerations (including knowledge gaps).
- A study design operationalization review which would be similar to code review for programmers but focused on best practices for observational retrospective clinical research studies
- Optimization of cohort definitions for improved run times
- Assistance with cohort characterization based on the target sources
- Assistance with feature extraction, including awareness of pre-build features and suggestions/guidance on development of custom features
- Assistance designing and executing patient level prediction model development workflows

Features:
- Tooling so that existing powerful OHDSI tools are used including WebAPI REST endpoints, output from DataQualityDashboard, Achilles aggregate output on the source clinical data, hooks and features in HADES libraries for extensions (e.g., custom features), JSON descriptions of cohorts, concept set, etc.
- The target models will be fine tuned (e.g., using LoRA) on the many previously completed OHDSI studies where a written higher level study concept has been turned into a detailed written protocol linking to serialized artifacts

such as JSON concept sets, cohort definitions, etc.

I would like you to critique this concept to start with. I am just now learning about ACP (e.g., from https://agentclientprotocol.com/overview/introduction) and this idea might be improved using a different method. However, I think from an architecture perspective, a single ACP server could be the right thing because, historically, there have been common computational artifacts between the no-code Atlas/WebAPI project and the R-based HADES toolstack. As you critique, include consideration of whether projects like Open-WebUI could be leveraged so that an OHDSI custom model (fine tuned as mentioned above) might work very well.

Love the direction. You're picking the right seams in the OHDSI stack (ATLAS/WebAPI + HADES) where an agent can actually add leverage. Below is a crisp critique with architectural options, pitfalls, and a staged plan.

Big picture critique

- **Value is there**:
    - *Concept set refinement* leveraging `concept_ancestor` / `concept_relationship`, plus source-specific utilization, is a concrete agent sweet spot. (You can ground suggestions in CDM vocab tables, and show diffs + rationales.) OHDSI +1
    - *Phenotype import/adaptation* from prior definitions into executable cohort JSON is realistic because ATLAS/WebAPI and the HADES packages already serialize artifacts in standard forms. OHDSI +1

  - *Study design "operationalization review"* maps to a linter/static-analysis vibe over ATLAS specs + HADES scripts with DQD/Achilles context (profiling the data, flagging poor overlap, etc.). OHDSI +1
  - *Runtime optimization of cohorts* is actionable: recommend index hints, simplify inclusion rules, remove never-hit criteria based on source characterization, and pre-materialize temp tables. Achilles/DQD give the signal; the agent proposes patches. OHDSI +1
  - *Feature extraction / PLP workflows*: HADES has well-defined touchpoints (FeatureExtraction, PatientLevelPrediction, CohortGenerator), so an agent can scaffold study packages, run diagnostics, and render comparisons. OHDSI +2
- **Your "single ACP server" instinct is reasonable—but which ACP?**
  There are (at least) two "ACPs" floating around now:
  1. **Agent Client Protocol (agentclientprotocol.com)** from Zed—focused on standardizing how IDEs talk to coding agents. Good if you want editors and web UIs to share a consistent contract for "chat, tool calls, edits." Agent Client Pr… +1
  2. **Agent Communication Protocol** (IBM/others) aimed at agent-to-agent interoperability. Useful if you anticipate a mesh of specialized agents (concept agent, phenotype agent, runtime agent) cooperating. IBM +1
     For OHDSI, the *editor-style* Agent Client Protocol is the cleaner fit to wire a "study-design-by-chat" panel into ATLAS and also into RStudio/VS Code. If later you split responsibilities across multiple agents, you can add the agent-to-agent ACP as an internal bus.
- **Open-WebUI is a pragmatic UI shell**: it gives you model-agnostic chat, function/tools, and customizable RAG without inventing a front-end. You can expose WebAPI/HADES calls as tools, plus run an OHDSI-tuned model (or route to

vendors/self-hosted). This can be embedded or proxied behind ATLAS with SSO.

Open WebUI

Architecture options (from “least invasive” to “platformy”)

1. **Tool-first, no new protocol (MVP)**
    - Keep ATLAS as-is; add a right-side “Assistant” panel that calls your backend service. The backend speaks **WebAPI** to ATLAS resources and shells out to **HADES R** via an adapter. The model uses function-calling to trigger:
        - Concept-set diff/proposals (reads vocab via source vocab schema or WebAPI).
        - Cohort JSON lint/patch proposals.
        - DQD/Achilles awareness to warn about low data support.
    - Pros: fastest path; pure REST + Rscript. Cons: not standardized for broader IDE/editor reuse. GitHub +3
2. **Agent Client Protocol (ACP) “editor contract”**
    - Implement your backend as an **ACP agent server**. The ATLAS “Assistant panel” and an R IDE extension (RStudio addin/VS Code) both talk ACP. Your agent exposes “tools” that map to WebAPI endpoints and HADES commands.
    - Pros: unified behavior across ATLAS and developer tools; swap models easily; future-proof multi-client. Cons: you’re aligning with a fast-moving community spec. Agent Client Pr… +1
3. **Agent mesh (Agent-to-Agent ACP)**
    - Split responsibilities: *Vocabulary Agent*, *Phenotype Agent*, *Runtime Agent*, *PLP Agent*, each with specific tools and memory. Useful at scale; more complex infra. IBM +1

4. **Open-WebUI as the hub**
    - Host Open-WebUI, register your OHDSI tools (WebAPI, DQD, Achilles, HADES adapters), mount a project knowledge base (prior protocols, cohort JSONs). Embed Open-WebUI inside ATLAS via an iframe or new tab with shared auth. Pros: rich admin UX + tool catalog; Cons: double-UI feel unless you style it tightly. Open WebUI

Critical design points (don't skip)

- **Data governance / PHI boundaries**
  Keep the agent blind to row-level PHI. Feed it: vocab tables, cohort JSON, aggregate Achilles outputs, DQD results, and model diagnostics. No raw patient data leaves the DBMS. (Achilles/DQD were designed to summarize safely—use that.) OHDSI +1
- **Reproducibility & provenance**
  Every suggestion should come with: (1) the exact SQL/JSON patch, (2) the evidence (counts, DQD checks, Achilles stats) and (3) a "why." Persist a machine-readable **"study advice log"** alongside the study package.
- **Human-in-the-loop by default**
  Only *propose* changes; show diffs and confidence + expected runtime impact. Allow one-click apply -> regenerates ATLAS spec or R study skeleton.
- **Model training reality check**
  Your "fine-tune on prior OHDSI studies" is promising—but mind licenses/provenance of public protocol text and whether serialized artifacts are consistently available. Start with **instruction-tuning** on (a) TheBookOfOHDSI excerpts, (b) redacted protocols you own, (c) synthetic study briefs -> proper cohort

JSON pairs. Use LoRA to keep infra light. (HADES/PLP docs give consistent targets for tool usage exemplars.) OHDSI

- **Scope your tool surface**
  Start with these tool bindings:
  - WebAPI: list/get/update concept sets & cohorts; run SQL via JDBC proxy only on vocab & temp schema; export cohort definitions. GitHub
  - HADES: `CohortGenerator` , `FeatureExtraction` , `PatientLevelPrediction` wrappers (containerized R with pinned versions); return JSON summaries/plots. OHDSI Forums +1
  - DQD/Achilles: read results JSON/DB tables for guardrails and guidance. OHDSI +1
- **Performance guidance that matters**
  Have the agent inspect cohort SQL (post-SqlRender) and suggest: reduce `JOIN` s on sparse domains, transform OR chains to temp tables, push filters before joins, and use concept rollups from `concept_ancestor` instead of enumerating large children lists when possible. Back the advice with Achilles prevalence numbers ("<50 records for concept X—drop or broaden?"). OHDSI +1

Where ACP helps (and where it doesn't)

- **Helps**: a stable, editor-like contract for chat messages, tool calls, file/patch proposals, streaming updates. This lets you implement **once** and reuse in ATLAS, RStudio, VS Code, and even terminals. Agent Client Pr...
- **Doesn't solve**: auth to WebAPI, OHDSI role/permission mapping, and enterprise secrets. You'll still need a service that trades ACP requests for (OIDC-backed) WebAPI calls.

Security & auth sketch

- SSO into ATLAS → short-lived token → your agent backend exchanges for a WebAPI token, stores per-project credentials in memory, and refuses raw table scans.
- For R/HADES runs, launch ephemeral containers with read-only mounts to study artifacts; write outputs to a controlled results bucket.

Suggested MVP (8–10 focused weeks of build time)

1. **Two “tools” only**
    - *Concept Set Advisor*: given a draft set + study intent, propose additions/removals using vocabulary relationships and Achilles prevalence; render a reasoned diff.
    - *Cohort Linter*: analyze a cohort JSON; highlight rules likely to cause empty/non-portable cohorts; propose rewrites + expected runtime changes using DQD/Achilles signals. OHDSI +1
2. **UI**
    - Add a panel in ATLAS (React) that speaks ACP to your backend (or start with Open-WebUI embedded). Open WebUI
3. **HADES adapter**
    - Containerize R with CohortGenerator/FeatureExtraction/PLP; expose a simple “run & summarize” endpoint with deterministic seeds and pinned versions. OHDSI Forums +1
4. **Eval**
    - **Concept sets**: compare agent vs expert Jaccard/F1 and downstream cohort counts.
    - **Cohort runtime**: measure wall-clock + I/O before/after.

  - **PLP**: time-to-first-validated-model and adherence to PLP best practices (train/val/test splits, leakage checks). GitHub

Alternative to ACP (if you want even less friction now)

- Use Open-WebUI with **Tools/Functions** to call WebAPI/HADES and drop it into ATLAS; swap to ACP later once your tool contract stabilizes. This gets you multi-model support, simple RAG, and a working admin UI out of the box. Open WebUI

Bottom line

- Your concept is sound: start **tool-first** with a thin service that knows WebAPI + HADES and reads DQD/Achilles outputs; keep the agent *advisory* and reproducible.
- ACP (editor-style) is a good bet if you want one agent that works in **ATLAS and R IDEs**. Start with Open-WebUI if you want speed and a flexible shell today; you can still standardize on ACP later. Agent Client Pr... +1

If you want, I'll sketch the initial tool schemas (inputs/outputs) for "Concept Set Advisor" and "Cohort Linter" so you can hand them to an engineer and start building.

Sources

Let's iterate once more after I answer some questions that came up with the feedback you gave me:

- You asked for which ACP - I was indeed thinking of the editor-style. Having something that an editor-like method for communicating with tools and a fine-tuned model across R and Atlas/WebAPI is fundamental to the community's culture.

R users will want a functional interface that works within RStudio and plays very nicely with R's REPL. Something like this within R:
-- > OHDSIAssistant::lintStudyDesign(studyProtocol="<url to a study protocol or path to one on disk>", studyPackage="<url to a github study package or '.' if it is in the current folder>", lintTasks=c("concept-sets-review","cohort-critique-general-design","cohort-critique-source-specific-concerns", ...)) # the function could then show thoughts and tool execution, request approval for certain tasks (e.g., agree to plan, create new files, modify existing files, or others), and summarize improvements with justification.

Atlas/WebAPI users will want an Assistant chat interface that allows them to point to specific concept sets, cohort definitions, characterizations, and sources that the agent will then use for analysis. The agent will suggest a plan, ask for approval or feedback, and then implement the plan by creating new concept sets, cohort definitions, an characterizations via WebAPI that the user can review in Atlas.

- I think that, in order to generate excitement and strong commitment to support the development of a fully mature implementation, having a study design operationalization review which would be similar to code review for programmers but focused on best practices for observational retrospective clinical research studies, and leveraging the incredible work of the OHDSI community (beyond just Book of OHDSI which is outdated), recent advances in causal inference, and a self critical/cautious vs speed-is-everything approach.

- Open-webui embedded is not an option but I would love to have the ACP agent use its API to access the models (e.g., one customized for this), tools, knowledge bases, etc that support these use cases.

This is headed the right way. With your clarifications, here's a focused iteration that turns the idea into something engineers (and the OHDSI community) can rally around—using an editor-style ACP, first-class R/ATLAS experiences, and Open-WebUI only as a back-end API.

North-star goals

- One **agent contract** (ACP) used from R (RStudio REPL) and from an ATLAS "Assistant" panel.
- Agent is **advisory + reproducible**: plans, diffs, justifications, and opt-in apply steps.
- Guardrails from **DQD/Achilles/CohortDiagnostics** + recent causal-inference best practices (target trial framing; bias checks) baked into a "study operationalization review."

---

1) Developer-facing contract: ACP-first

Core ACP tool suite (minimal, but high leverage)

- `get_artifact(ref)` → fetches concept set / cohort / characterization by `ref` (WebAPI id or file path/URL).
- `propose_concept_set_diff({conceptSetRef, studyIntent, sourceStatsRef})` → JSON Patch + rationale.
- `cohort_lint({cohortRef, sourceStatsRef, dqdRef})` → findings (severity, evidence, fixes).
- `cohort_optimize({cohortRef, sourceStatsRef})` → SQL rewrite suggestions + expected runtime deltas.
- `characterize({cohortRef, sourceRef, templateRef})` → runs characterization and returns summaries.
- `feature_advise({targetRef, outcomeRef, sourceRef})` → prebuilt vs custom feature suggestions.
- `plp_scaffold({targets, outcomes, covariateSettings, design})` → PLP package skeleton + README.
- `plp_train_eval({packageRef, sourceRef})` → training summary, diagnostics, leakage checks.

Each tool returns:

- `plan` (what/why), `evidence` (counts/diagnostics pointers), `patches` (JSONPatch/SQL diffs), and `risk_notes` (assumptions/limitations).

---

2) R experience: package sketch

Your proposed call is spot-on. Here's a realistic surface plus the ACP wiring:

r

Copy code

```r
# OHDSIAssistant R package: user-facing API
OHDSIAssistant::lintStudyDesign(
  studyProtocol = "path/or/url/to/protocol.md",
  studyPackage  = ".",
  lintTasks     = c(
    "concept-sets-review",
    "cohort-critique-general-design",
    "cohort-critique-source-specific",
    "immortal-time-bias",
    "positivity/overlap",
    "temporal-alignment",
    "leakage-checks"
  ),
  sourceStatsRef = "achilles://cdm_x",
  dqdRef         = "dqd://cdm_x",
  apply          = FALSE,           # require user approval
  interactive    = TRUE,            # REPL prompts; show step logs
  streamThoughts = TRUE             # stream ACP messages in console
)
```

How it behaves

- Resolves refs (WebAPI ids, local files, GitHub URLs).
- Calls ACP tools; streams **plan** → **evidence** → **patch**.
- For each patch, prompts: **[a]pply / [s]kip / [v]iew-diff**.
- Writes a machine-readable **advice log** under `inst/assistant/` (YAML + JSONPatch + links to evidence).
- Provides `OHDSIAssistant::applyPatches()` and `OHDSIAssistant::revertPatches()` with a git-friendly workflow.

Internals

- `acp_client <- OHDSIAssistant::acp_connect(url, token)` (configurable).
- WebAPI and HADES calls live server-side; the R package remains thin.

---

3) ATLAS/WebAPI experience

ATLAS “Assistant” panel

- **Target selectors**: Concept set(s), Cohort(s), Characterization, Source(s).
- **Action presets**: “Study Review,” “Concept Set Review,” “Cohort Optimization,” “PLP Scaffold.”
- Agent posts a **plan with checkboxes**; user approves steps.
- On apply, the agent creates **new resources** (e.g., `*-assistant-v1`) via WebAPI so users can diff and adopt or discard.

WebAPI integration

- Read-only for most actions; write creates *new* versions/resources (never in-place edits).
- Auth via your existing ATLAS SSO; backend trades short-lived tokens.

---

4) “Study operationalization review”: what it actually checks

Think of this as a linter + reviewer, with rule packs you can grow:

Design framing

- Target trial language present? Target/outcome anchors? Time-at-risk windows aligned with mechanism?
- Inclusion/exclusion coherence; conflicting rules; criteria that never hit in source.

Bias/validity

- Immortal time bias risk (index vs exposure ascertainment).
- Positivity/overlap flags using source prevalence (Achilles) and cohort diagnostics.
- Leakage checks for PLP (post-index features sneaking into train).
- Confounding by indication hints; suggests negative/positive controls.

Transportability & feasibility

- Rare concept usage; cross-source portability concerns.
- Missingness/sparsity signals from Achilles/DQD.
- ICD/SNOMED/LOINC domain mismatches; unit normalization reminders.

Performance/run-time

- SQL anti-patterns (OR chains, filters after joins, repeated concept lists).
- Proposes: temp tables for subqueries, early filters, ancestor roll-ups instead of enumerating many children.

Each finding reports: **severity**, **impact (design/validity/perf)**, **evidence ref**, **suggested fix**.

---

5) How Open-WebUI fits (without embedding)

- Treat Open-WebUI as a **model/router API** behind your ACP server:
    - Define model routes (e.g., `ohdsi-tuned-lora`) and **tool adapters** that simply forward to your ACP tools.

  - Your ACP server authenticates to Open-WebUI once; clients never see it.
- Benefits: swap models, add vector-stores/KBs, log traces centrally.
- Keep the UI in ATLAS and R; Open-WebUI is your **model/tool runtime**, not the face.

---

6) Provenance, safety, and privacy

- **No PHI** leaves the DBMS. The agent consumes:
  - Vocab tables; cohort JSON; WebAPI-exported specs.
  - **Aggregate** outputs: Achilles, DQD, CohortDiagnostics summaries.
- Every suggestion is **traceable**:
  - Store `plan.md` , `rationale.json` , `patch.json` , and evidence links (e.g., `achilles://…` , `webapi://conceptSet/123`).
- **Apply** steps always fork artifacts (suffix with `-assistant-v{n}`) and add ATLAS description notes with the plan summary.

---

7) Model & knowledge strategy

- Start with **instruction-tuning** + high-quality exemplars you control:
  - (a) Your own protocols + artifacts (redacted).
  - (b) Synthetic pairs (brief → cohort JSON/PLP skeleton) generated and then filtered by validators.
  - (c) Public OHDSI docs/guides (for tool usage, not proprietary datasets).
- LoRA adapters per sub-task are fine (concept set advising vs cohort linting vs PLP scaffolding).

- Retrieval: index prior study skeletons, concept sets, and “good fixes” so the agent cites precedents.

---

8) Concrete schemas (hand-to-engineer)

Example: `cohort_lint` request/response

json

Copy code

```json
// request
{
  "cohortRef": "webapi://cohort/456",
  "sourceStatsRef": "achilles://cdm_x",
  "dqdRef": "dqd://cdm_x",
  "rules": ["immortal-time-bias","positivity","temporal-alignment","sql-an
}
```

json

Copy code

```json
// response (trimmed)
{
  "plan": "Review cohort 456 for design risks and runtime issues.",
  "findings": [
    {
      "id": "immortal_time_risk",
      "severity": "high",
      "impact": "validity",
      "evidence": [{"ref": "achilles://cdm_x/domain/observation_period", "
      "proposal": {"type": "text", "content": "Anchor index on first expos
    },
    {
      "id": "sql_or_chain",
      "severity": "medium",
      "impact": "performance",
      "evidence": [{"ref": "sql://rendered", "note": "6 ORs across visits"
      "proposal": {"type": "sql_patch", "diff": "<unified-diff>"}
    }
  ],
  "patches": [
    {"artifact": "webapi://cohort/456", "type": "jsonpatch", "ops": [ /* …
  ],
  "risk_notes": ["Portability may suffer on sources lacking X." ]
}
```

Example: `propose_concept_set_diff`

Inputs: `conceptSetRef` , `studyIntent` , optional `sourceStatsRef` .
Outputs: added/removed conceptIds with **why** (ancestor/descendant links, utilization, domain mismatch), JSONPatch for ATLAS concept set, and a one-click “create new concept set” instruction.

---

9) Minimal roadmap (4 tight milestones)

1. **M0 – Wire the pipes (ACP skeleton + two tools)**
    - ACP server + auth

   - Tools: `propose_concept_set_diff`, `cohort_lint` (subset: runtime + empty criteria)
   - R package: `lintStudyDesign()` minimal; interactive approvals
   - ATLAS panel: select artifacts, run review, show plan/diffs (create `*-assistant-v1`)
2. **M1 – Source-aware guardrails**
   - Ingest Achilles + DQD; show linked evidence in findings
   - Add `cohort_optimize` and `characterize` (reads a standard template)
3. **M2 – PLP assist**
   - `plp_scaffold + plp_train_eval` via containerized HADES
   - Leakage/temporal checks, baseline performance dashboard
4. **M3 – Phenotype import/adapt**
   - Import common eMERGE/PhenotypeLibrary JSON and propose ATLAS cohort criteria with caveats

---

10) Risks & how you blunt them

- **Spec churn (ACP / tools)** → keep a thin compatibility layer; version your tool contracts (`v1alpha`, `v1`).
- **Community trust** → default to *non-destructive forks*, verbose rationale, and side-by-side diffs.
- **Model hallucinations** → every suggestion must be accompanied by machine-verifiable evidence or a “speculative” flag that disables “apply.”

---

11) What to build next (concrete)

- Define the first **10 lint rules** with inputs, detection logic, and fix templates.
- Lock the **R function signatures** (as above) and generate a pkg skeleton.
- Draft the **ATLAS panel** interaction (plan → approve → apply) and the naming/versioning scheme for forked artifacts.
- Pick the **two tools** for M0 and write their JSON schemas (I can draft those if you want code-ready OpenAPI).

If you'd like, I'll produce (a) the OpenAPI/JSON schemas for `propose_concept_set_diff` and `cohort_lint`, (b) the R package scaffold (DESCRIPTION, minimal functions, roxygen), and (c) the ATLAS panel wireframe copy—so you can hand this to a team and start sprint 1.

Nice! btw, I am the engineer and the person driving the concept so I don't mind you showing technial details.
Let's proceed as follows:

Preliminary:
- I am using openai codex to write and edit code. I have created a new github repo that is completely empty save a README.md
- The environment the codex tool runs in has a fully functional R environment with Hades as long as R is started with the .renv activated

Task:
Develop a prompt that I can give to codex to build out

two lint rules ("concept-sets-review", "cohort-critique-general-design") within the new package folder. For now we will build a basic prototype that will simulate having a fine-tuned model by using the ACP version of the Gemini CLI (https://github.com/google-gemini/gemini-cli) I will specify an API key to use. Have codex build both the ACP adaptation and R package for the prototype so that I can demonstrate the concept within an interactive R session.

Awesome—here's a single, copy-paste **Codex build prompt** you can drop into your empty repo. It tells Codex exactly what files to create, how they should work together, and how to demo the two rules end-to-end in an interactive R session using **Gemini CLI (ACP mode)** as the stand-in "model." It includes a local fallback so you can demo even if Gemini isn't running yet.

🔧 Paste this into Codex

Goal: Scaffold a minimal but working prototype that (a) exposes two lint rules via an ACP-style adapter, (b) provides an R package `OHDSIAssistant` with `lintStudyDesign()` that calls those tools, and (c) uses Gemini CLI ACP as the "model" behind the adapter (with a no-ML fallback).

High-level plan

- Create a mono-repo with:
  - `acp/` — a tiny HTTP bridge that looks like an ACP "agent server" to the R client and optionally proxies prompts to **Gemini CLI (ACP)**.

  - `R/OHDSIAssistant/` — an R package that exposes `lintStudyDesign()` and friends, connects to the ACP bridge, and prints a human-readable plan/findings/patches.
  - `demo/` — a toy “study” with a cohort JSON and concept set JSON files to lint against.
  - `scripts/` — helper launchers.
- Implement **two lint rules**:
  1. `concept-sets-review` — take a concept set JSON, flag obvious problems: empty set, mixed domains, or concepts without valid ancestors (using `concept_id` metadata if present); returns a JSON Patch proposal that (for demo) removes obviously broken entries and adds a rationale.
  2. `cohort-critique-general-design` — take a cohort JSON and check: missing washout, missing time-at-risk, inconsistent inclusion rule ordering, and impossible criteria (e.g., date windows inverted). Returns structured findings and a patch suggestion (e.g., add a default 365d washout).
- Provide **Gemini path** (optional): if `GEMINI_ACP_ENABLED=1`, the bridge will send a succinct prompt+context to `gemini` CLI and merge the LLM’s suggestions with deterministic checks. Otherwise, it only uses deterministic checks.
- Preserve **provenance**: include `plan`, `findings[]`, `patches[]`, and `risk_notes[]` in responses.

---

0) Create repository layout

Create the following files:

pgsql

Copy code

```
.
├─ acp/
│  ├─ server.py
│  ├─ tool_schemas/
│  │  ├─ propose_concept_set_diff.schema.json
│  │  └─ cohort_lint.schema.json
│  ├─ README.md
├─ R/
│  └─ OHDSIAssistant/
│     ├─ DESCRIPTION
│     ├─ NAMESPACE
│     ├─ R/
│     │  ├─ acp_client.R
│     │  ├─ lintStudyDesign.R
│     │  ├─ utils_json.R
│     │  └─ rules_local_fallback.R
│     ├─ inst/
│     │  └─ assistant/.gitkeep
│     └─ man/.gitkeep
├─ demo/
│  ├─ concept_set.json
│  ├─ cohort_definition.json
│  └─ protocol.md
├─ scripts/
│  ├─ start_acp.sh
│  ├─ start_gemini_acp_example.sh
│  └─ r_demo_commands.R
├─ .gitignore
├─ README.md
```

---

1) `acp/server.py` — minimal ACP-ish bridge (HTTP JSON)

Write a small Python 3 server (Flask or FastAPI). It exposes:

- `GET /health → {status:"ok"}`
- `POST /tools/propose_concept_set_diff` → implements **concept-sets-review** logic:
  - Input: `{ "conceptSetRef": "<path-or-url>", "studyIntent": "<text>", "sourceStatsRef": "<optional>" }`
  - Loads JSON from file path or http(s).
  - Deterministic checks: empty set, mixed domains (if `domain_id` present), duplicated concept_ids.
  - Output: `{ plan, findings[], patches[], risk_notes[] }`
- `POST /tools/cohort_lint` → implements **cohort-critique-general-design** logic:
  - Input: `{ "cohortRef": "<path-or-url>" }`
  - Deterministic checks: missing washout/time-at-risk, inverted dates, empty inclusion rules, impossible windows.
  - Output same shape.

If `os.getenv("GEMINI_ACP_ENABLED") == "1"`, both endpoints:

- Build a short **LLM prompt** with the artifact excerpts (truncate to ~3–5KB).
- Call Gemini CLI synchronously:
  - `subprocess.run(["gemini","chat","--model", os.getenv("GEMINI_MODEL","gemini-2.0-flash-exp"), "--input", prompt], capture_output=True, check=True)`
- Parse LLM suggestions if they match a simple JSON schema: `{ "findings": [...], "patches": [...] }`. If parsing fails, ignore and continue with

deterministic output.

- Merge LLM results with deterministic ones (deduplicate by finding id).

Implement helper utilities:

- `load_json(ref)` : file path or https URL
- `json_patch_for_conceptset_removals(add=[], remove=[])`
- `simple_patch_note(text)`

Return consistent JSON.

Code (create full file):

python Copy code

```python
# acp/server.py
import os, json, subprocess, sys
from urllib.parse import urlparse
import requests
from flask import Flask, request, jsonify

app = Flask(__name__)

def load_json(ref: str):
    if ref.startswith("http://") or ref.startswith("https://"):
        r = requests.get(ref, timeout=30)
        r.raise_for_status()
        return r.json()
    with open(ref, "r", encoding="utf-8") as f:
        return json.load(f)

def maybe_call_gemini(prompt: str):
    if os.getenv("GEMINI_ACP_ENABLED") != "1":
        return None
    model = os.getenv("GEMINI_MODEL", "gemini-2.0-flash-exp")
    try:
        p = subprocess.run(
            ["gemini", "chat", "--model", model, "--input", prompt],
            capture_output=True, check=True, text=True
        )
        txt = p.stdout.strip()
        # Try to find a JSON blob in the output
        start = txt.find("{")
        end = txt.rfind("}")
        if start != -1 and end != -1 and end > start:
            return json.loads(txt[start:end+1])
    except Exception as e:
        print(f"[gemini-warning] {e}", file=sys.stderr)
    return None

@app.get("/health")
def health():
    return jsonify({"status":"ok"})

@app.post("/tools/propose_concept_set_diff")
def propose_concept_set_diff():
    body = request.get_json(force=True)
    ref = body.get("conceptSetRef")
    study_intent = body.get("studyIntent","")
    cs = load_json(ref)

    # Expect ATLAS-like shape, but be defensive:
```

```
# Allow either { items:[{concept:{conceptId, domainId}}...] } or a sim
items = []
if isinstance(cs, dict) and "items" in cs:
    for it in cs.get("items", []):
        c = it.get("concept") or {}
        items.append({"conceptId": c.get("conceptId"), "domainId": c.g
elif isinstance(cs, list):
    for it in cs:
        items.append({"conceptId": it.get("conceptId"), "domainId": it

findings = []
patches = []
risk_notes = []
plan = f"Review concept set for gaps and inconsistencies given the stu

# Deterministic checks
concept_ids = [x.get("conceptId") for x in items if x.get("conceptId")
unique_ids = set(concept_ids)
duplicates = [cid for cid in concept_ids if concept_ids.count(cid) > 1
if len(items) == 0:
    findings.append({"id":"empty_concept_set","severity":"high","impac
if duplicates:
    findings.append({"id":"duplicate_concepts","severity":"medium","im

# Mixed domains detection if domainId present
domains = set([x.get("domainId") for x in items if x.get("domainId")])
if len(domains) > 1:
    findings.append({"id":"mixed_domains","severity":"low","impact":"p

# Demo patch: remove duplicates
```

```
    if duplicates:
        ops = []
        # naive: just note removals; in real ATLAS we would target items b
        ops.append({"op":"note","path":"/items","value":{"removeDuplicates
        patches.append({"artifact": ref, "type":"jsonpatch", "ops": ops})

    # Optional Gemini merge
    prompt = f"""You are checking an OHDSI concept set for study design is
Study intent: {study_intent}
First 20 items: {json.dumps(items[:20])}
Return JSON with fields: findings[], patches[]. Keep patches high-level no
"""
    llm = maybe_call_gemini(prompt)
    if llm:
        # Merge simply
        for f in llm.get("findings", []):
            if f not in findings: findings.append(f)
        for p in llm.get("patches", []):
            if p not in patches: patches.append(p)

    return jsonify({
        "plan": plan,
        "findings": findings,
        "patches": patches,
        "risk_notes": risk_notes
    })

@app.post("/tools/cohort_lint")
def cohort_lint():
    body = request.get_json(force=True)
```

```
ref = body.get("cohortRef")
cohort = load_json(ref)

plan = "Review cohort JSON for general design issues (washout/time-at-
findings, patches, risk_notes = [], [], []

# Extremely simplified detection across common ATLAS exports
washout = None
tar = None
try:
    pc = cohort.get("PrimaryCriteria", {})
    washout = pc.get("ObservationWindow", {})
    tar = cohort.get("TargetOutcome", cohort.get("CensoringCriteria", 
except Exception:
    pass

if not washout or washout.get("PriorDays") in (None, 0):
    findings.append({"id":"missing_washout","severity":"medium","impac
    patches.append({"artifact":ref,"type":"jsonpatch","ops":[{"op":"no

# Fake check for inverted dates in inclusion rules
try:
    irules = cohort.get("InclusionRules", [])
    for i, r in enumerate(irules):
        # demo heuristic: if rule has a 'start' > 'end' in any window-
        w = r.get("window", {})
        if w and w.get("start",0) > w.get("end",0):
            findings.append({"id":f"inverted_window_{i}","severity":"h
except Exception:
    pass
    # Optional Gemini merge
    prompt = f"""You are reviewing an OHDSI cohort JSON for general design
Return JSON fields findings[], patches[] (patches are notes, not executabl
Cohort excerpt: {json.dumps({k:cohort.get(k) for k in list(cohort.keys())[
"""
    llm = maybe_call_gemini(prompt)
    if lll := llm:
        for f in lll.get("findings", []):
            if f not in findings: findings.append(f)
        for p in lll.get("patches", []):
            if p not in patches: patches.append(p)

    return jsonify({
        "plan": plan,
        "findings": findings,
        "patches": patches,
        "risk_notes": risk_notes
    })

if __name__ == "__main__":
    port = int(os.getenv("ACP_PORT", "7777"))
    app.run(host="127.0.0.1", port=port)
```

Dependencies: add to `acp/README.md` :

md Copy code

```md
# ACP bridge
- Python 3.9+
- pip install flask requests
- Optional: Gemini CLI (ACP) installed and authenticated
  - export GEMINI_ACP_ENABLED=1
  - export GEMINI_MODEL=gemini-2.0-flash-exp
```

Create minimal tool schemas as documentation (not enforced):

json Copy code

```json
// acp/tool_schemas/propose_concept_set_diff.schema.json
{
  "title": "propose_concept_set_diff",
  "type": "object",
  "properties": {
    "conceptSetRef": {"type":"string"},
    "studyIntent": {"type":"string"},
    "sourceStatsRef": {"type":"string"}
  },
  "required": ["conceptSetRef"]
}
```

json Copy code

```json
// acp/tool_schemas/cohort_lint.schema.json
{
  "title": "cohort_lint",
  "type": "object",
  "properties": {
    "cohortRef": {"type":"string"}
  },
  "required": ["cohortRef"]
}
```

---

2) R package skeleton

DESCRIPTION

text Copy code

```text
Package: OHDSIAssistant
Type: Package
Title: Assistant Prototype for OHDSI Study Design Linting
Version: 0.0.1
Authors@R: person("Your","Name", email="you@example.org", role=c("aut","cr
Description: Prototype assistant client that connects to a local ACP bridg
License: MIT + file LICENSE
Encoding: UTF-8
Depends: R (>= 4.2)
Imports: jsonlite, httr, utils
Suggests:
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.2
```

NAMESPACE

r Copy code

```r
export(acp_connect)
export(lintStudyDesign)
```

R/acp_client.R

```r
acp_state <- new.env(parent = emptyenv())

#' Connect to ACP bridge
#' @param url e.g. "http://127.0.0.1:7777"
#' @param token optional bearer
#' @return invisible(TRUE)
acp_connect <- function(url = "http://127.0.0.1:7777", token = NULL) {
  acp_state$url <- sub("/$", "", url)
  acp_state$token <- token
  resp <- httr::GET(paste0(acp_state$url, "/health"))
  if (httr::status_code(resp) != 200) stop("ACP bridge not reachable")
  invisible(TRUE)
}

.acp_post <- function(path, body) {
  url <- paste0(acp_state$url, path)
  headers <- list(`Content-Type`="application/json")
  if (!is.null(acp_state$token)) headers$Authorization <- paste("Bearer", 
  resp <- httr::POST(url, body = jsonlite::toJSON(body, auto_unbox = TRUE)
  if (httr::status_code(resp) >= 300) stop("ACP error: ", httr::content(re
  jsonlite::fromJSON(httr::content(resp, as="text"), simplifyVector = FALS
}
```

R/utils_json.R

r

Copy code

```r
read_json_ref <- function(ref) {
  # For demo, just read local files; URLs allowed.
  if (grepl("^https?://", ref)) {
    txt <- readLines(ref, warn = FALSE)
    return(jsonlite::fromJSON(paste(txt, collapse="\n"), simplifyVector = 
  }
  jsonlite::fromJSON(ref, simplifyVector = FALSE)
}

print_findings <- function(findings) {
  if (length(findings) == 0) {
    cat("  ✓ No findings.\n")
    return(invisible(NULL))
  }
  for (f in findings) {
    cat(sprintf("  - [%s][%s] %s\n",
                toupper(f$severity %||% "INFO"),
                f$impact %||% "",
                f$message %||% jsonlite::toJSON(f, auto_unbox=TRUE)))
  }
}

`%||%` <- function(a,b) if (is.null(a)) b else a
```

R/rules_local_fallback.R

r

Copy code

```
# Local deterministic checks used if ACP bridge is not connected (or as pr
local_concept_sets_review <- function(conceptSetRef, studyIntent="") {
  cs <- read_json_ref(conceptSetRef)
  items <- if (!is.null(cs$items)) cs$items else cs

  get_item <- function(it) {
    c <- it$concept %||% it
    list(conceptId = c$conceptId %||% c$id %||% NA_integer_,
         domainId = c$domainId %||% NA_character_)
  }
  list_items <- lapply(items, get_item)

  plan <- sprintf("Local concept set review for %s", conceptSetRef)
  findings <- list(); patches <- list(); risk_notes <- list()

  ids <- vapply(list_items, function(x) x$conceptId, integer(1))
  ids <- ids[!is.na(ids)]
  if (length(list_items) == 0) {
    findings <- c(findings, list(list(id="empty_concept_set", severity="hi
  }
  if (length(ids)) {
    dups <- ids[duplicated(ids)]
    if (length(dups)) {
      findings <- c(findings, list(list(id="duplicate_concepts", severity=
      patches <- c(patches, list(list(artifact=conceptSetRef, type="jsonpa
    }
  }
  domains <- unique(vapply(list_items, function(x) x$domainId %||% NA_chara
  domains <- domains[!is.na(domains)]
  if (length(domains) > 1) {
    findings <- c(findings, list(list(id="mixed_domains", severity="low", 
  }

  list(plan=plan, findings=findings, patches=patches, risk_notes=risk_note
}

local_cohort_critique_general <- function(cohortRef) {
  cdef <- read_json_ref(cohortRef)
  plan <- sprintf("Local general cohort design lint for %s", cohortRef)
  findings <- list(); patches <- list(); risk_notes <- list()

  pc <- cdef$PrimaryCriteria %||% list()
  wash <- pc$ObservationWindow %||% list()
  if (is.null(wash$PriorDays) || identical(wash$PriorDays, 0L)) {
    findings <- c(findings, list(list(id="missing_washout", severity="medi
    patches <- c(patches, list(list(artifact=cohortRef, type="jsonpatch", 
  }

  irules <- cdef$InclusionRules %||% list()
  for (i in seq_along(irules)) {
    w <- irules[[i]]$window %||% NULL
    if (!is.null(w) && !is.null(w$start) && !is.null(w$end) && w$start > w
      findings <- c(findings, list(list(id=paste0("inverted_window_", i), 
    }
  }
  list(plan=plan, findings=findings, patches=patches, risk_notes=risk_note
}
```

R/lintStudyDesign.R

Copy code

r

Copy code

```r
#' Lint study design (prototype)
#' @param studyProtocol path or URL to protocol.md
#' @param studyPackage path to local study folder ('.' ok)
#' @param lintTasks character vector of tasks
#' @param apply logical; not used in prototype (advisory only)
#' @param interactive logical; print plans and findings
#' @param streamThoughts logical; no-op placeholder
#' @return list of results by task
lintStudyDesign <- function(
  studyProtocol,
  studyPackage = ".",
  lintTasks = c("concept-sets-review","cohort-critique-general-design"),
  apply = FALSE,
  interactive = TRUE,
  streamThoughts = TRUE
) {
  # Resolve demo refs:
  conceptSetRef <- file.path(studyPackage, "concept_set.json")
  cohortRef     <- file.path(studyPackage, "cohort_definition.json")

  results <- list()
  use_acp <- !is.null(acp_state$url)

  if ("concept-sets-review" %in% lintTasks) {
    if (use_acp) {
      res <- .acp_post("/tools/propose_concept_set_diff", list(
        conceptSetRef = conceptSetRef,
        studyIntent = paste(readLines(studyProtocol, warn = FALSE), collap
      ))
    } else {
      res <- local_concept_sets_review(conceptSetRef, studyIntent = paste(
    }
    if (interactive) {
      cat("\n== Concept Sets Review ==\n")
      cat(res$plan, "\n")
      print_findings(res$findings)
    }
    results$`concept-sets-review` <- res
  }

  if ("cohort-critique-general-design" %in% lintTasks) {
    if (use_acp) {
      res <- .acp_post("/tools/cohort_lint", list(
        cohortRef = cohortRef
      ))
    } else {
      res <- local_cohort_critique_general(cohortRef)
    }
    if (interactive) {
      cat("\n== Cohort Critique: General Design ==\n")
      cat(res$plan, "\n")
      print_findings(res$findings)
    }
    results$`cohort-critique-general-design` <- res
  }

  # Write advice log
  outdir <- file.path(studyPackage, "inst", "assistant")
  if (!dir.exists(outdir)) dir.create(outdir, recursive = TRUE)
```

```
  ts <- format(Sys.time(), "%Y%m%dT%H%M%S")
  jsonlite::write_json(results, file.path(outdir, paste0("advice_", ts, ".

  invisible(results)
}
```

---

3) Demo assets

demo/concept_set.json (simple mixed-domain, duplicate)

json Copy code

```json
{
  "items": [
    { "concept": { "conceptId": 111, "domainId": "Condition" } },
    { "concept": { "conceptId": 222, "domainId": "Drug" } },
    { "concept": { "conceptId": 111, "domainId": "Condition" } }
  ]
}
```

demo/cohort_definition.json (missing washout, one inverted rule)

json Copy code

```json
{
  "PrimaryCriteria": {
    "ObservationWindow": { "PriorDays": 0, "PostDays": 0 }
  },
  "InclusionRules": [
    { "name": "ok" },
    { "name": "bad_window", "window": { "start": 10, "end": 5 } }
  ]
}
```

demo/protocol.md

```md
Study intent: Evaluate safety outcomes for exposure X in adults with condi
Key risks: prevalent-user bias; improper time-at-risk; limited overlap.
```

---

4) Scripts

scripts/start_acp.sh

```bash
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")/.."
export FLASK_ENV=production
export ACP_PORT=${ACP_PORT:-7777}
# Set these to enable Gemini
# export GEMINI_ACP_ENABLED=1
# export GEMINI_MODEL=gemini-2.0-flash-exp
python3 -m pip install --quiet flask requests
python3 acp/server.py
```

```
chmod +x scripts/start_acp.sh
```

scripts/start_gemini_acp_example.sh

bash Copy code

```bash
#!/usr/bin/env bash
# Helper that checks Gemini CLI is installed and authenticated.
set -euo pipefail
if ! command -v gemini >/dev/null 2>&1; then
  echo "Gemini CLI not found. Install it first."
  exit 1
fi
export GEMINI_ACP_ENABLED=1
export GEMINI_MODEL=${GEMINI_MODEL:-gemini-2.0-flash-exp}
echo "Gemini ACP enabled with model $GEMINI_MODEL"
```

```
chmod +x scripts/start_gemini_acp_example.sh
```

scripts/r_demo_commands.R

r Copy code

```r
# Run from repo root in an R session after starting the ACP bridge:
# source("scripts/r_demo_commands.R")

devtools::load_all("R/OHDSIAssistant") # or build & install
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")
res <- OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","cohort-critique-general-design"
  apply         = FALSE,
  interactive   = TRUE
)
str(res, max.level = 2)
```

---

5) Root files

.gitignore

bash Copy code

```bash
.Rproj.user
.Rhistory
.RData
.Ruserdata
.Rbuildignore
.Rproj
.DS_Store
**/inst/assistant/*.json
```

README.md

md Copy code

```md
# OHDSI Assistant Prototype

This repo contains:
- A tiny **ACP-like bridge** (`acp/server.py`) that exposes two "tools":
```

~~~
  - `/tools/propose_concept_set_diff` (concept-sets-review)
  - `/tools/cohort_lint` (cohort-critique-general-design)
- An **R package** `OHDSIAssistant` with `lintStudyDesign()` that calls th
- A **demo** study in `demo/`.

## Quick start

1) Start bridge:

```bash
./scripts/start_acp.sh
~~~

(Optional) enable Gemini CLI (ACP-like):

```bash
./scripts/start_gemini_acp_example.sh  # sets env vars
./scripts/start_acp.sh
```

2. In R:

```r
devtools::load_all("R/OHDSIAssistant")
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")
OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","cohort-critique-general-design"
  apply         = FALSE,
```

```
  interactive   = TRUE
)
```

You’ll see plans, findings, and suggested patches. Advice logs are saved under `demo/inst/assistant/`.

yaml

```yaml
---

### 6) Acceptance checks (Codex: run these mentally / as comments)

- `./scripts/start_acp.sh` runs and `GET /health` returns `{"status":"ok"}
- With the bridge running, in R:
  - `acp_connect()` succeeds.
  - `lintStudyDesign()` prints:
    - In **concept-sets-review**: flags duplicates and mixed domains; emit
    - In **cohort-critique-general-design**: flags missing washout and inv
  - A JSON advice file appears in `demo/inst/assistant/`.
- If `GEMINI_ACP_ENABLED=1` and `gemini` is available, outputs may include

---

### 7) Notes for future extension (leave as comments)

- Replace “note” patches with real **JSON Patch** targeting ATLAS export s
- Add WebAPI integration: accept `webapi://cohort/<id>` and resolve to JSO
- Add Achilles/DQD readers to enrich evidence and severity levels.
```

```
---

**End of prompt**
```

Regenerate the prompt you want me to give to codes to be a single text/markdown block so I can more easily paste it.

markdown

Copy code

```markdown
# Codex Build Prompt — OHDSI Assistant Prototype (ACP + R)

**Goal:** Create a minimal, working prototype that (a) exposes two lint ru

---

## Repo layout to create
```

```
.
├─ acp/
│  ├─ server.py
│  ├─ tool_schemas/
│  │  ├─ propose_concept_set_diff.schema.json
│  │  └─ cohort_lint.schema.json
│ ├── README.md
├── R/
│ └── OHDSIAssistant/
│ ├── DESCRIPTION
│ ├── NAMESPACE
│ ├── R/
│ │ ├── acp_client.R
│ │ ├── lintStudyDesign.R
│ │ ├── utils_json.R
│ │ └── rules_local_fallback.R
│ ├── inst/
│ │ └── assistant/.gitkeep
│ └── man/.gitkeep
├── demo/
│ ├── concept_set.json
│ ├── cohort_definition.json
│ └── protocol.md
├── scripts/
│ ├── start_acp.sh
│ ├── start_gemini_acp_example.sh
│ └── r_demo_commands.R
├── .gitignore
├── README.md
```

python

Copy code

## Files & contents

### `acp/server.py` (Flask bridge; deterministic checks + optional Gemini)

```python
import os, json, subprocess, sys
import requests
from flask import Flask, request, jsonify

app = Flask(__name__)

def load_json(ref: str):
    if ref.startswith("http://") or ref.startswith("https://"):
        r = requests.get(ref, timeout=30)
        r.raise_for_status()
        return r.json()
    with open(ref, "r", encoding="utf-8") as f:
        return json.load(f)

def maybe_call_gemini(prompt: str):
    if os.getenv("GEMINI_ACP_ENABLED") != "1":
        return None
    model = os.getenv("GEMINI_MODEL", "gemini-2.0-flash-exp")
    try:
        p = subprocess.run(
            ["gemini", "chat", "--model", model, "--input", prompt],
            capture_output=True, check=True, text=True
        )
        txt = p.stdout.strip()
        start = txt.find("{")
        end = txt.rfind("}")
        if start != -1 and end != -1 and end > start:
            return json.loads(txt[start:end+1])
    except Exception as e:
        print(f"[gemini-warning] {e}", file=sys.stderr)
    return None

@app.get("/health")
def health():
    return jsonify({"status":"ok"})

@app.post("/tools/propose_concept_set_diff")
def propose_concept_set_diff():
    body = request.get_json(force=True)
    ref = body.get("conceptSetRef")
    study_intent = body.get("studyIntent","")
    cs = load_json(ref)

    items = []
    if isinstance(cs, dict) and "items" in cs:
        for it in cs.get("items", []):
            c = it.get("concept") or {}
            items.append({"conceptId": c.get("conceptId"), "domainId": c.g
    elif isinstance(cs, list):
        for it in cs:
            items.append({"conceptId": it.get("conceptId"), "domainId": it

    findings = []
    patches = []
    risk_notes = []
    plan = f"Review concept set for gaps and inconsistencies given the stu

    concept_ids = [x.get("conceptId") for x in items if x.get("conceptId")
    duplicates = [cid for cid in concept_ids if concept_ids.count(cid) > 1
    if len(items) == 0:
        findings.append({"id":"empty_concept_set","severity":"high","impac
    if duplicates:
        findings.append({"id":"duplicate_concepts","severity":"medium","im

    domains = set([x.get("domainId") for x in items if x.get("domainId")])
    if len(domains) > 1:
        findings.append({"id":"mixed_domains","severity":"low","impact":"p

    if duplicates:
        ops = [{"op":"note","path":"/items","value":{"removeDuplicatesOf":
        patches.append({"artifact": ref, "type":"jsonpatch", "ops": ops})

    prompt = f"""You are checking an OHDSI concept set for study design is
Study intent: {study_intent}
First 20 items: {json.dumps(items[:20])}
Return JSON with fields: findings[], patches[]. Keep patches high-level no
"""
    llm = maybe_call_gemini(prompt)
    if llm:
        for f in llm.get("findings", []):
            if f not in findings: findings.append(f)
        for p in llm.get("patches", []):
            if p not in patches: patches.append(p)
    return jsonify({"plan": plan, "findings": findings, "patches": patches

@app.post("/tools/cohort_lint")
def cohort_lint():
    body = request.get_json(force=True)
    ref = body.get("cohortRef")
    cohort = load_json(ref)

    plan = "Review cohort JSON for general design issues (washout/time-at-
    findings, patches, risk_notes = [], [], []

    pc = cohort.get("PrimaryCriteria", {}) if isinstance(cohort, dict) els
    washout = pc.get("ObservationWindow", {})

    if not washout or washout.get("PriorDays") in (None, 0):
        findings.append({"id":"missing_washout","severity":"medium","impac
        patches.append({"artifact":ref,"type":"jsonpatch","ops":[{"op":"no

    try:
        irules = cohort.get("InclusionRules", [])
        for i, r in enumerate(irules):
            w = r.get("window", {})
            if w and w.get("start",0) > w.get("end",0):
                findings.append({"id":f"inverted_window_{i}","severity":"h
    except Exception:
        pass

    prompt = f"""You are reviewing an OHDSI cohort JSON for general design
Return JSON fields findings[], patches[] (patches are notes, not executabl
Cohort excerpt: {json.dumps({k:cohort.get(k) for k in list(cohort.keys())[
"""
    llm = maybe_call_gemini(prompt)
    if llm:
        for f in llm.get("findings", []):
            if f not in findings: findings.append(f)
        for p in llm.get("patches", []):
            if p not in patches: patches.append(p)

    return jsonify({"plan": plan, "findings": findings, "patches": patches

if __name__ == "__main__":
    port = int(os.getenv("ACP_PORT", "7777"))
    from flask import Flask
    app.run(host="127.0.0.1", port=port)
```

acp/tool_schemas/propose_concept_set_diff.schema.json

json

Copy code

```json
{
  "title": "propose_concept_set_diff",
  "type": "object",
  "properties": {
    "conceptSetRef": {"type":"string"},
    "studyIntent": {"type":"string"},
    "sourceStatsRef": {"type":"string"}
  },
  "required": ["conceptSetRef"]
}
```

acp/tool_schemas/cohort_lint.schema.json

json Copy code

```json
{
  "title": "cohort_lint",
  "type": "object",
  "properties": {
    "cohortRef": {"type":"string"}
  },
  "required": ["cohortRef"]
}
```

acp/README.md

md Copy code

```md
# ACP Bridge (Prototype)

- Python 3.9+
- Install deps: `python3 -m pip install flask requests`
- Optional Gemini CLI:
  - `export GEMINI_ACP_ENABLED=1`
  - `export GEMINI_MODEL=gemini-2.0-flash-exp`
- Endpoints:
  - `GET /health`
  - `POST /tools/propose_concept_set_diff`
  - `POST /tools/cohort_lint`
```

R/OHDSIAssistant/DESCRIPTION

```text
Package: OHDSIAssistant
Type: Package
Title: Assistant Prototype for OHDSI Study Design Linting
Version: 0.0.1
Authors@R: person("Your","Name", email="you@example.org", role=c("aut","cr
Description: Prototype assistant client that connects to a local ACP bridg
License: MIT + file LICENSE
Encoding: UTF-8
Depends: R (>= 4.2)
Imports: jsonlite, httr, utils
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.2
```

R/OHDSIAssistant/NAMESPACE

```r
export(acp_connect)
export(lintStudyDesign)
```

R/OHDSIAssistant/R/acp_client.R

r Copy code

```r
acp_state <- new.env(parent = emptyenv())

#' Connect to ACP bridge
#' @param url e.g. "http://127.0.0.1:7777"
#' @param token optional bearer token
#' @return invisible(TRUE)
acp_connect <- function(url = "http://127.0.0.1:7777", token = NULL) {
  acp_state$url <- sub("/$", "", url)
  acp_state$token <- token
  resp <- httr::GET(paste0(acp_state$url, "/health"))
  if (httr::status_code(resp) != 200) stop("ACP bridge not reachable")
  invisible(TRUE)
}

.acp_post <- function(path, body) {
  if (is.null(acp_state$url)) stop("ACP not connected; call acp_connect().
  url <- paste0(acp_state$url, path)
  headers <- list(`Content-Type`="application/json")
  if (!is.null(acp_state$token)) headers$Authorization <- paste("Bearer",
  resp <- httr::POST(url, body = jsonlite::toJSON(body, auto_unbox = TRUE)
                     httr::add_headers(.headers = headers))
  if (httr::status_code(resp) >= 300) stop("ACP error: ", httr::content(re
  jsonlite::fromJSON(httr::content(resp, as="text"), simplifyVector = FALS
}
```

R/OHDSIAssistant/R/utils_json.R

```r
read_json_ref <- function(ref) {
  if (grepl("^https?://", ref)) {
    txt <- readLines(ref, warn = FALSE)
    return(jsonlite::fromJSON(paste(txt, collapse="\n"), simplifyVector = 
  }
  jsonlite::fromJSON(ref, simplifyVector = FALSE)
}

print_findings <- function(findings) {
  if (length(findings) == 0) {
    cat("  ✓ No findings.\n"); return(invisible(NULL))
  }
  for (f in findings) {
    cat(sprintf("  - [%s][%s] %s\n",
                toupper((f$severity %||% "INFO")),
                (f$impact %||% ""),
                (f$message %||% jsonlite::toJSON(f, auto_unbox=TRUE))))
  }
}

`%||%` <- function(a,b) if (is.null(a)) b else a
```

R/OHDSIAssistant/R/rules_local_fallback.R

```r
```

```
local_concept_sets_review <- function(conceptSetRef, studyIntent="") {
  cs <- read_json_ref(conceptSetRef)
  items <- if (!is.null(cs$items)) cs$items else cs

  get_item <- function(it) {
    c <- it$concept %||% it
    list(conceptId = c$conceptId %||% c$id %||% NA_integer_,
         domainId  = c$domainId %||% NA_character_)
  }
  lst <- lapply(items, get_item)

  plan <- sprintf("Local concept set review for %s", conceptSetRef)
  findings <- list(); patches <- list(); risk_notes <- list()

  ids <- vapply(lst, function(x) x$conceptId, integer(1))
  ids <- ids[!is.na(ids)]
  if (length(lst) == 0) {
    findings <- c(findings, list(list(id="empty_concept_set", severity="hi
  }
  if (length(ids)) {
    dups <- ids[duplicated(ids)]
    if (length(dups)) {
      findings <- c(findings, list(list(id="duplicate_concepts", severity=
                                        message=paste("Duplicate conceptId
      patches <- c(patches, list(list(artifact=conceptSetRef, type="jsonpa
                                      ops=list(list(op="note", path="/item
    }
  }
  domains <- unique(vapply(lst, function(x) x$domainId %||% NA_character_,
  domains <- domains[!is.na(domains)]
  if (length(domains) > 1) {
    findings <- c(findings, list(list(id="mixed_domains", severity="low", 
                                      message=paste("Multiple domains:", p
  }

  list(plan=plan, findings=findings, patches=patches, risk_notes=risk_note
}

local_cohort_critique_general <- function(cohortRef) {
  cdef <- read_json_ref(cohortRef)
  plan <- sprintf("Local general cohort design lint for %s", cohortRef)
  findings <- list(); patches <- list(); risk_notes <- list()

  pc <- cdef$PrimaryCriteria %||% list()
  wash <- pc$ObservationWindow %||% list()
  if (is.null(wash$PriorDays) || identical(wash$PriorDays, 0L)) {
    findings <- c(findings, list(list(id="missing_washout", severity="medi
                                      message="No or zero-day washout; con
    patches <- c(patches, list(list(artifact=cohortRef, type="jsonpatch",
                                    ops=list(list(op="note", path="/Primar
                                                  value=list(ProposedPrior
  }

  irules <- cdef$InclusionRules %||% list()
  for (i in seq_along(irules)) {
    w <- irules[[i]]$window %||% NULL
    if (!is.null(w) && !is.null(w$start) && !is.null(w$end) && w$start > w
      findings <- c(findings, list(list(id=paste0("inverted_window_", i), 
                                        message=sprintf("InclusionRules[%d
    }
  }
  list(plan=plan, findings=findings, patches=patches, risk_notes=risk_note
}
```

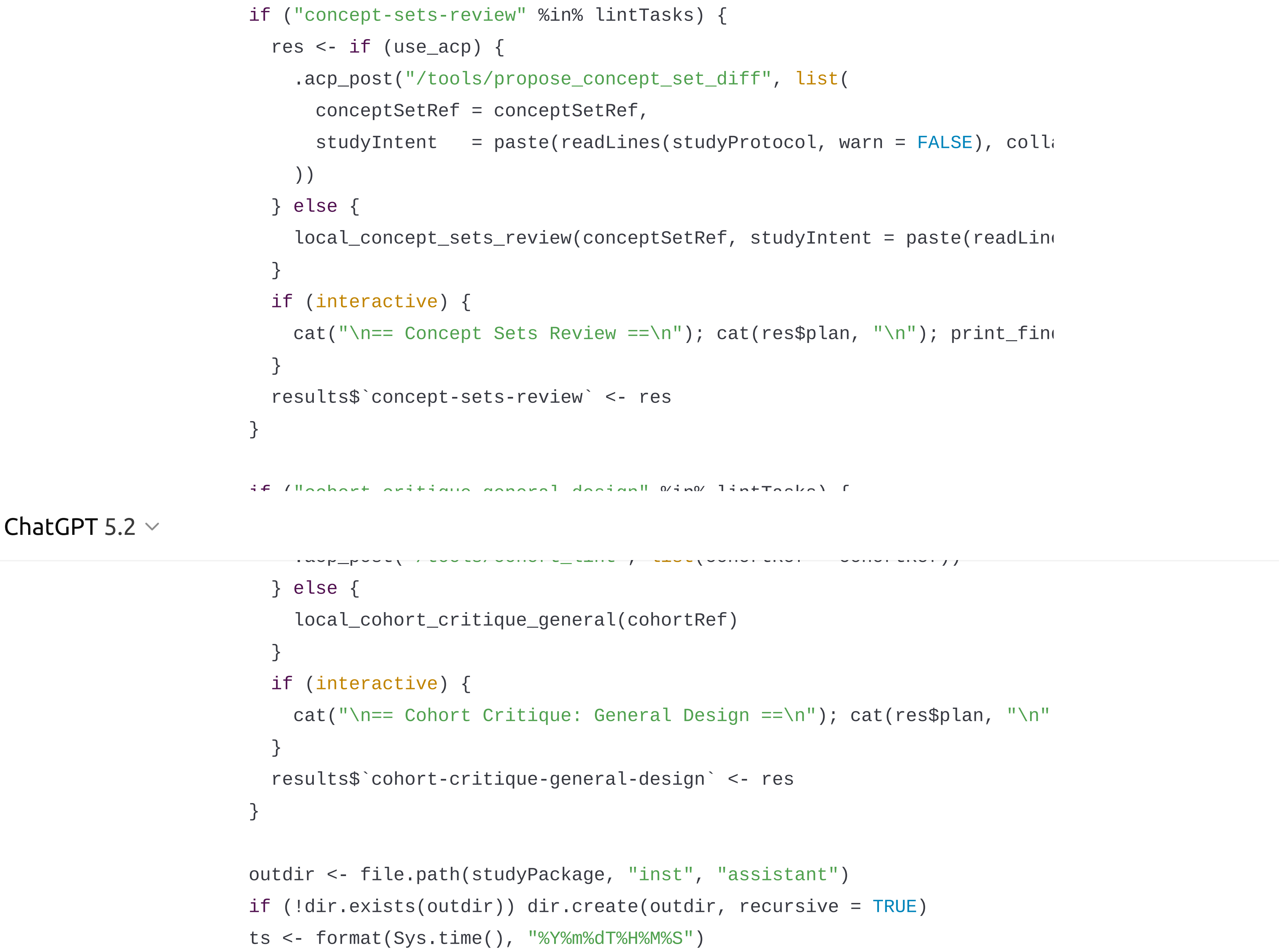
R/OHDSIAssistant/R/lintStudyDesign.R

r

Copy code

```r
#' Lint study design (prototype)
#' @param studyProtocol path or URL to protocol.md
#' @param studyPackage path to local study folder ('.' ok)
#' @param lintTasks character vector of tasks
#' @param apply logical; not used in prototype (advisory only)
#' @param interactive logical; print plans and findings
#' @param streamThoughts logical; placeholder
#' @return list of results by task
lintStudyDesign <- function(
  studyProtocol,
  studyPackage = ".",
  lintTasks = c("concept-sets-review","cohort-critique-general-design"),
  apply = FALSE,
  interactive = TRUE,
  streamThoughts = TRUE
) {
  conceptSetRef <- file.path(studyPackage, "concept_set.json")
  cohortRef     <- file.path(studyPackage, "cohort_definition.json")

  results <- list()
  use_acp <- !is.null(acp_state$url)
```



```
if ("concept-sets-review" %in% lintTasks) {
  res <- if (use_acp) {
    .acp_post("/tools/propose_concept_set_diff", list(
      conceptSetRef = conceptSetRef,
      studyIntent   = paste(readLines(studyProtocol, warn = FALSE), coll
    ))
  } else {
    local_concept_sets_review(conceptSetRef, studyIntent = paste(readLin
  }
  if (interactive) {
    cat("\n== Concept Sets Review ==\n"); cat(res$plan, "\n"); print_fin
  }
  results$`concept-sets-review` <- res
}

if ("cohort-critique-general-design" %in% lintTasks) {
    .acp_post("/tools/cohort_lint", list(cohortRef = cohortRef))
  } else {
    local_cohort_critique_general(cohortRef)
  }
  if (interactive) {
    cat("\n== Cohort Critique: General Design ==\n"); cat(res$plan, "\n"
  }
  results$`cohort-critique-general-design` <- res
}

outdir <- file.path(studyPackage, "inst", "assistant")
if (!dir.exists(outdir)) dir.create(outdir, recursive = TRUE)
ts <- format(Sys.time(), "%Y%m%dT%H%M%S")
```

```
  jsonlite::write_json(results, file.path(outdir, paste0("advice_", ts, ".

  invisible(results)
}
```

Demo inputs

`demo/concept_set.json`

json Copy code

```json
{
  "items": [
    { "concept": { "conceptId": 111, "domainId": "Condition" } },
    { "concept": { "conceptId": 222, "domainId": "Drug" } },
    { "concept": { "conceptId": 111, "domainId": "Condition" } }
  ]
}
```

`demo/cohort_definition.json`

json Copy code

```json
{
  "PrimaryCriteria": {
    "ObservationWindow": { "PriorDays": 0, "PostDays": 0 }
  },
  "InclusionRules": [
    { "name": "ok" },
    { "name": "bad_window", "window": { "start": 10, "end": 5 } }
  ]
}
```

`demo/protocol.md`

```md
Study intent: Evaluate safety outcomes for exposure X in adults with condi
Key risks: prevalent-user bias; improper time-at-risk; limited overlap.
```

Scripts

`scripts/start_acp.sh`

```bash
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")/.."
export FLASK_ENV=production
export ACP_PORT=${ACP_PORT:-7777}
python3 -m pip install --quiet flask requests
python3 acp/server.py
```

`scripts/start_gemini_acp_example.sh`

```bash
```

```
#!/usr/bin/env bash
set -euo pipefail
if ! command -v gemini >/dev/null 2>&1; then
  echo "Gemini CLI not found. Install & auth it first."
  exit 1
fi
export GEMINI_ACP_ENABLED=1
export GEMINI_MODEL=${GEMINI_MODEL:-gemini-2.0-flash-exp}
echo "Gemini ACP enabled with model $GEMINI_MODEL"
```

scripts/r_demo_commands.R

r

Copy code

```r
# Run after starting the ACP bridge:
# source("scripts/r_demo_commands.R")

devtools::load_all("R/OHDSIAssistant")  # or build & install
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")
res <- OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","cohort-critique-general-design"
  apply         = FALSE,
  interactive   = TRUE
)
str(res, max.level = 2)
```

Root files

`.gitignore`

bash Copy code

```bash
.Rproj.user
.Rhistory
.RData
.Ruserdata
.Rbuildignore
.Rproj
.DS_Store
**/inst/assistant/*.json
__pycache__/
```

`README.md`

md Copy code

```md
# OHDSI Assistant Prototype

This repo contains:
- A tiny **ACP-like bridge** (`acp/server.py`) exposing:
  - `/tools/propose_concept_set_diff` (concept-sets-review)
  - `/tools/cohort_lint` (cohort-critique-general-design)
- An **R package** `OHDSIAssistant` with `lintStudyDesign()` that calls th
- A **demo** study in `demo/`.

## Quick start
```

~~~
1) Start bridge:

```bash
./scripts/start_acp.sh
~~~

(Optional) Enable Gemini CLI:

```bash
./scripts/start_gemini_acp_example.sh  # sets env vars
./scripts/start_acp.sh
```

2. In R:

```r
devtools::load_all("R/OHDSIAssistant")
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")
OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","cohort-critique-general-design"
  apply         = FALSE,
  interactive   = TRUE
)
```

You'll see plans, findings, and suggested patches. Advice logs are saved under `demo/inst/assistant/`.

Next steps

- Replace “note” patches with executable JSON Patch aligned to ATLAS schema.
- Add WebAPI resolvers for `webapi://cohort/<id>` and `webapi://conceptSet/<id>`.
- Pull in Achilles/DQD summaries to drive severity and evidence links.

yaml Copy code

```yaml


---
```

Ok. We are now to the point of testing the ACP client and R demo using a workspace in open-webui with an open source LLM as the remote model. The open source LLM needs to be prompted to reply in JSON to the ACP client so that the client can parse its suggested actions. Generate a system prompt that I will set in the workspace for the LLM that will enable this functionality.

markdown Copy code

```markdown
You are the **OHDSI Assistant (ACP Model)**. You must produce **only valid

## PURPOSE
Given a short prompt containing either a **concept set** excerpt or a **col
- `concept-sets-review`
- `cohort-critique-general-design`
```

~~~
## STRICT OUTPUT CONTRACT
Return a single JSON object with exactly these top-level keys:
- `plan`: string (≤ 300 chars) describing your review focus.
- `findings`: array of Finding objects (may be empty).
- `patches`: array of Patch objects (may be empty).
- `risk_notes`: array of strings with caveats/assumptions (may be empty).

### Finding object
```json
{
  "id": "snake_case_identifier",
  "severity": "low|medium|high",
  "impact": "design|validity|portability|performance",
  "message": "Concise human-readable finding (≤ 220 chars).",
  "evidence": [
    { "ref": "string (e.g., 'conceptId:123' or 'path:/PrimaryCriteria/Obse
  ]
}
~~~

Patch object (advisory; NOT executable edits)

Use one of the two forms below. Prefer **jsonpatch** with an advisory `note` op.

A) Advisory JSON Patch (notes only)

json Copy code

```json
{
  "artifact": "string echoing the referenced artifact",
  "type": "jsonpatch",
```

```
  "ops": [
    { "op": "note", "path": "/path/within/artifact", "value": { "summary":
  ]
}
```

B) Advice blob

json Copy code

```
{
  "artifact": "string echoing the referenced artifact",
  "type": "advice",
  "content": "Concise suggested change (≤ 600 chars)"
}
```

SCOPE & GUARDRAILS

- **Never invent IDs or structures** that are not present in the provided excerpt. You may reference observed fields/ids and generic paths (e.g., `/PrimaryCriteria/ObservationWindow`).
- **No PHI** handling; assume all inputs are aggregates/specs.
- If uncertain, **lower severity** and add a short explanation in `risk_notes`.
- If nothing to report, return empty arrays for `findings` and `patches` with a neutral `plan`.

HEURISTICS

For `concept-sets-review`

Look for:

- **Empty concept set** (`empty_concept_set` , severity: high, impact: design).
- **Duplicate conceptIds** (`duplicate_concepts` , medium, design).
- **Mixed domains** when that appears inconsistent or unexplained (`mixed_domains` , low, portability). Do not assert required domains unless explicitly indicated by the study intent excerpt.
- Optionally flag items missing expected fields (e.g., no `conceptId`) as (`malformed_item` , medium, design).

Patches (advisory):

- Add a `jsonpatch` `note` suggesting removal of duplicates, or separate concept sets by domain if appropriate.

For `cohort-critique-general-design`

Look for:

- **Missing or zero-day washout** in `/PrimaryCriteria/ObservationWindow.PriorDays` (`missing_washout` , medium, validity).
- **Inverted windows** in inclusion rules, e.g., `start > end` (`inverted_window_<index>` , high, validity).
- **Empty or contradictory inclusion rules** (`weak_inclusion_rules` , low/medium, design).
- **Ambiguous time-at-risk** if clearly absent in the excerpt (`unclear_time_at_risk` , low, validity).

Patches (advisory):

- `jsonpatch` `note` proposing a typical washout (e.g., 365 days) or clarifying window alignment.

STYLE & SIZE LIMITS

- Keep total JSON under **15 KB**.
- Strings must use **double quotes**; escape any embedded quotes.
- Do **not** include trailing commas or comments.

EXAMPLES (ILLUSTRATIVE; DO NOT EMIT THIS TEXT)

Example minimal success response:

json

Copy code

```json
{
  "plan": "Reviewed concept set for emptiness, duplicates, and domain mixi
  "findings": [
    {
      "id": "duplicate_concepts",
      "severity": "medium",
      "impact": "design",
      "message": "Duplicate conceptIds detected: [111].",
      "evidence": [{ "ref": "conceptId:111", "note": "Appears multiple tim
    }
  ],
  "patches": [
    {
      "artifact": "concept_set_ref",
      "type": "jsonpatch",
      "ops": [
        { "op": "note", "path": "/items", "value": { "summary": "Remove du
      ]
    }
  ],
  "risk_notes": []
}
```

—End of system instructions—

makefile Copy code

```makefile
::contentReference[oaicite:0]{index=0}
```

Good. In Open-webui I am setting up Gemma3 12B as the test model with the system prompt you generated. Here are parameter options that Open-webui could pass to the model. Many of these might not apply to Gemma. Examine them and search for experiences of others and gemma 3 docs to determine if any of these parameters should be set a certain way:

Advanced Params

Hide
Stream Chat Response

Default
Stream Delta Chunk Size

Default
Function Calling

Default
Seed

Default
Stop Sequence

Default
Temperature

Default
Reasoning Effort

Default
logit_bias

Default
max_tokens

Default
top_k

Default
top_p

Default
min_p

Default
frequency_penalty

Default
presence_penalty

Default
mirostat

Default
mirostat_eta

Default
mirostat_tau

Default
repeat_last_n

Default
tfs_z

Default
repeat_penalty

Default
use_mmap

Default
use_mlock

Default
think (Ollama)

Default
format (Ollama)

Default
num_keep (Ollama)

Default
num_ctx (Ollama)

Default
num_batch (Ollama)

Default
num_thread (Ollama)

Default
num_gpu (Ollama)

Default
keep_alive (Ollama)

Default

Here’s a **practical recommendation for the inference parameters** you should set in Open-WebUI when running **Gemma3-12B** as your remote model for structured JSON outputs to drive your ACP client.

Most community experience and recommendations for **Gemma 3** (including the 12B variant) point to a *sampling configuration* that balances coherence and diversity, which is useful when you want the model to *follow your structured system prompt and produce valid JSON* while still being responsive enough to generate enough content.

Hugging Face +1

✅ Recommended Inference Parameters for Gemma3-12B

Temperature: `1.0`

- Encourages responsive, diverse generation while still being reasonable for structured outputs.
- Lower temperatures (e.g., `0.2–0.5`) would make generation more deterministic, but can sometimes lead to overly terse JSON or repetition, *especially with complex outputs like ACP JSON.*
- Setting it to `1.0` aligns with community usage recommended for Gemma and maintains more natural token sampling. Hugging Face +1

Top_p: `0.95`

- Helps avoid overly greedy responses while still keeping the token space restricted to the top 95% of total probability mass.
- For structured formats, top_p around 0.9–0.95 is commonly recommended to balance completeness and correctness. Hugging Face

Top_k: `64`

- Restricts sampling to the top 64 probable tokens at every step.
- This is a common Gemma setting that works well with top_p and temperature for structured generation without degenerate tokens. Hugging Face

Repetition_penalty: `1.0` (default / off)

- Repetition penalty generally *isn’t necessary* for structured JSON tasks, because the JSON format itself constrains output structure.
- You can leave this at default unless you observe looping or degenerate repeated outputs. Unsloth Docs

Min_p: `0.01` or leave at default

- If your interface supports `min_p`, setting it very low (e.g., `0.01`) can help ensure the model considers less probable but still valid tokens needed for exact JSON syntax, though this parameter is optional. Unsloth Docs

Max_tokens: Set sufficiently high for JSON

- Since your structured JSON responses can be long, ensure `max_tokens` is large enough (e.g., 2048–4096) so that the model doesn’t cut off mid-JSON. The exact max depends on how much detail you expect in plans/findings/patches.

🛠️ Recommended Settings Summary (for Open-WebUI)

Parameter	Value	Purpose
temperature	`1.0`	balanced sampling for correctness + sufficient variation
top_p	`0.95`	restricts output to focused, high-prob tokens
top_k	`64`	token selection constraint useful for structured output
repetition_penalty	`1.0`	no extra repetition bias

Parameter	Value	Purpose
min_p *(optional)*	`0.01`	allows tokens outside the strict top_p tail if needed
max_tokens	`2048–4096`	enough budget for JSON structures
stream chat response	*off* (recommended)	because you want complete JSON per message
stop sequence	best with JSON end	define a closing brace `}` stop if your client supports it

---

🧠 Why These Settings

1) Structured JSON needs balanced sampling

- Too low temperature makes the model *too deterministic* and can lead to short or incomplete JSON or badly parsed JSON arrays.
- Too high top_p or very high temperature can lead to random tokens and break JSON syntax.
- The commonly recommended combo of `temp=1.0`, `top_p=0.95`, `top_k=64` is a stable middle ground that many users employ for structured JSON tasks. Hugging Face

2) Gemma defaults are reasonable

- Community inference guidelines for Gemma3 variants generally cite temperature ~1.0 with top_p ~0.95 and top_k ~64 as a good baseline. Unsloth Docs

3) Stopping and streaming

- For structured JSON generation, it's usually better to **turn off streaming** so you get one *complete JSON document* per response; streaming can complicate parsing partial JSON.
- Define a stop sequence if your UI allows it (like `}` plus newline) so the model avoids extraneous tokens after the JSON object end.

⚠️ Parameters that *don't* matter much for this use-case

- **logit_bias** – Useful only for very fine control; not needed initially.
- **presence_penalty / frequency_penalty** – Mostly relevant for creative text; minimally useful for structured JSON.
- **mirostat, mirostat_tau, mirostat_eta** – Sampling algorithms mainly affecting creative text, unlikely to help stable JSON generation.
- **Seed** – Useful if you want reproducible sampling; set same seed for determinism across runs.
- **repeat_last_n, tfs_z** – Internal sampling tweaks with limited utility for JSON tasks.

🧪 If you *need more determinism*

- If the model occasionally produces malformed JSON, try lowering **temperature toward 0.2–0.3**, and keep **top_p ≤ 0.9**.
- Or set **temperature = 0.0** to enforce greedy decoding (highest-prob tokens only), but this can sometimes truncate structured output or make it overly fixed.
- Start with the baseline above and tweak only if you see issues in JSON completion.

🧩 Example Open-WebUI Config (copy this)

```yaml
temperature: 1.0
top_p: 0.95
top_k: 64
min_p: 0.01
repetition_penalty: 1.0
max_tokens: 4096
stream_chat_response: false
```

This configuration maximizes JSON correctness + full structured outputs for your ACP client.

If you want, I can also draft a simple **reusable template** for prompting Gemma in Open-WebUI that wraps your system prompt and these parameters into a cohesive generation request that's optimized for JSON output.

Sources

That worked pretty well. To give you an example of the state of the prototype, I have pasted below some logging output from the ACP client based on the R session also pasted below. Take no action yet, just review this and store it in your context window for future reference.

ACP client interaction with Gemma3 in the open-webui workspace:

```
127.0.0.1 - - [22/Dec/2025 04:36:21] "POST /
OPENWEBUI OUTGOING TEXT > You are checking a
OPENWEBUI OUTGOING TEXT > Study intent: Stud
OPENWEBUI OUTGOING TEXT > First 20 items: [{
OPENWEBUI OUTGOING TEXT > Return JSON with t
OPENWEBUI INCOMING RAW > {"id":"gemma3:12b-3
```

json\n{\n \"plan\": \"Review concept set for drug-related concepts, focusing on descendant inclusion for potential clinical drug codes, and addressing potential time-at-risk concerns given study intent.\",\n \"findings\": [\n   {\n   \"id\": \"suggest_descendants_concept_set\",\n   \"severity\": \"medium\",\n   \"impact\": \"design\",\n   \"message\": \"The concept set contains several drug concepts. Consider including descendants to capture formulations and clinical drug codes.\",\n   \"evidence\": [\n   {\n   \"ref\": \"conceptId:757688\",\n   \"note\": \"Appears as the first concept and is within the Drug domain. Consider adding includeDescendants: true for robustness in cohort definition.\"\n   }\n   ]\n   }\n],\n \"patches\": [\n   {\n   \"artifact\": \"concept_set_ref\",\n   \"type\": \"jsonpatch\",\n   \"ops\": [\n   {\n   \"op\": \"note\",\n   \"path\": \"/items\",\n   \"value\": {\n   \"summary\": \"Include descendants for drug concepts\",\n   \"details\": \"For each drug conceptId in the concept set, consider adding 'includeDescendants: true' to encompass clinical drug codes.\"\n }\n }\n]\n }\n],\n \"risk_notes\": []\n}\n

```
"}}],"object":"chat.completion","usage":{"re
127.0.0.1 - - [22/Dec/2025 05:11:14] "POST /
OPENWEBUI OUTGOING TEXT > You are reviewing
OPENWEBUI OUTGOING TEXT > Return JSON fields
OPENWEBUI OUTGOING TEXT > Cohort excerpt: {"
OPENWEBUI INCOMING RAW > {"id":"gemma3:12b-2
```

json\n{\n \"findings\": [\n {\n \"field\": \"Cohort.ConceptSets[0].expression.items[0].concept.VOCABULARY_ID\",\n \"severity\": \"Info\",\n \"description\": \"SNOMED is broad and might include stages of Parkinson's. Consider a more specific SNOMED subset or a combination of SNOMED and RxNorm concepts for more accurate cohort definition. The current definition might include patients with pre-symptomatic stages.\",\n \"risk\": \"Potential for over-inclusion\",\n \"recommendation\": \"Review SNOMED subset for Parkinson's. Consult with clinicians to determine appropriate level of specificity. Consider incorporating RxNorm for clinical assessment if needed.\"\n },\n {\n \"field\": \"Cohort.ConceptSets[2].expression.items\",\n \"severity\": \"Warning\",\n \"description\": \"Mixing Brand Name and Ingredient concepts (e.g., Sinemet, Stalevo, Xadago) within the same `Codeset` can lead to unexpected results. Patients taking generics of these medications might be excluded. Also, the concept 'Sinemet' has non-standard vocabulary ID.\",\n \"risk\": \"Potential for under-inclusion and inaccurate representation of drug exposure.\",\n \"recommendation\": \"Replace brand names with ingredient concepts (Carbidopa/Levodopa for Sinemet, etc.) to ensure complete representation of drug exposure.\"\n },\n {\n \"field\": \"Cohort.ConceptSets[3].expression.items\",\n \"severity\": \"Info\",\n \"description\": \"Similar to the Parkinson's drugs, the Neuroleptic Drugs `Codeset` also mixes brand names and ingredients. See note above for consequences.\",\n \"risk\": \"Potential for under-inclusion and inaccurate representation of drug exposure.\",\n \"recommendation\": \"Replace brand names with ingredient concepts. This is particularly important when dealing with generics or medications with different brand names in different regions.\"\n },\n {\n \"field\": \"Cohort.InclusionRules[0].expression.Criteria[0].Criteria.DrugExposure.CodesetId\",\n \"severity\": \"Info\",\n \"description\": \"The Exclusion rule uses a `CodesetId` referencing `Neuroleptic meds`. This might exclude individuals who have been prescribed a neuroleptic for another condition. Ensure that this is the intended behavior. If this exclusion is intended, it would be helpful to document in the cohort description.\",\n \"risk\": \"Potential for unintended exclusion of patients.\",\n \"recommendation\": \"Evaluate the exclusion criteria's impact on the patient population. Document the rationale for exclusion in the cohort description.\"\n },\n {\n \"field\": \"Cohort.PrimaryCriteria.ObservationWindow\",\n

```
\"severity\": \"Info\",\n      \"description\": \"Consider documenting why 180 days prior/post is used for the ObservationWindow. This aids in reproducibility and understanding the cohort's clinical context.\",\n      \"risk\": \"Lack of clarity on cohort construction logic.\",\n      \"recommendation\": \"Document the rationale for the chosen observation window duration in the cohort description.\"\n    }\n  ],\n  \"patch_recommendations\": [\n    \"Replace Brand Name drug concepts with ingredient concepts.\",\n    \"Review and refine SNOMED concepts for specific Parkinson's diagnosis criteria.\",\n    \"Document reasoning behind exclusion rules and observation windows.\"\n  ]\n}\n
"}}],"object":"chat.completion","usage":{"r
127.0.0.1 - - [22/Dec/2025 05:11:35] "POST
```

R session demonstration:

```
> devtools::load_all("R/OHDSIAssistant")
ℹ Loading OHDSIAssistant
> OHDSIAssistant::acp_connect("http://127.0
> OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","
  apply         = FALSE,
  interactive   = TRUE
)
> OHDSIAssistant::lintStudyDesign(
  studyProtocol = "demo/protocol.md",
  studyPackage  = "demo",
  lintTasks     = c("concept-sets-review","
  apply         = FALSE,
  interactive   = TRUE
)
No encoding supplied: defaulting to UTF-8.

== Concept Sets Review ==
Review concept set for gaps and inconsisten
  - [MEDIUM][design] The concept set contai
No encoding supplied: defaulting to UTF-8.

== Cohort Critique: General Design ==
Review cohort JSON for general design issue
  - [INFO][] {"description":"SNOMED is broa
  - [WARNING][] {"description":"Mixing Bran
  - [INFO][] {"description":"Similar to the
  - [INFO][] {"description":"The Exclusion
  - [INFO][] {"description":"Consider docum
```

Got it—read everything and I've saved the details from the ACP ↔ Gemma3 logs and your R session output for reference. Ready when you want to proceed.

The prototype currently is only making suggestions. To be really useful, it will need to have actions that it can take with the user's approval. In the case of R session, these would be modifications to file-based artifacts that can be loaded into R such as concept sets, cohort definitions. Also, changes to R code used to execute the varous parts of study design. For Atlas/WebAPI, this would be modifications to concept sets and cohort definition objects passed from the Atlas client to the WebAPI in REST calls, with side effects on record stored in the `concept_set` and `cohort_*` tables that WebAPI uses to track these artifacts. Let's start with the simple task of patching a concept set JSON to use "includeDescendants:" true`. The heurisitic that the Gemma model's system prompt has is as follows:`

"

For `concept-sets-review`

- test: *Failure to use descendant concepts in the DRUG domain when possibly relevant* (`suggest_descendants_concept_set`, severity: medium, impact: design)
    - example: concepts in the DRUG domain where the concept set uses only an ingredient concept without specifying `"includeDescendants:" true`
    - patch: Add a `jsonpatch` note explaining to the user that the concept set would likely be sub-optimal if used in a cohort definition with drug_exposure criteria because record in that table tend to have clinical drug codes (i.e., codes that specify drug concepts with strengh and formulation). An appropriate action would be to edit the concept set to set all ingredient concepts in the JSON concept set definition to have

```
"includeDescendants:" true"
"
```

Develop a plan for how I would add tooling, both on the open-webui and the ACP sides, to make editing possible within R. Include in the design consideration for how to work with Atlas/WebAPI in the near future.

Love it — moving from "advice" to "edits with approval" is the inflection point where this becomes truly useful. Here's a concrete, engineer-ready plan that lets you patch artifacts in R now, while cleanly paving the road to ATLAS/WebAPI updates soon.

1) Editing model: advisory → actionable → applied

Introduce two layers of patching

1. **Advisory patches** (what your LLM already returns): human-readable "notes".
2. **Executable actions** (deterministic): a compact, schema-aware operation your ACP server understands and can turn into exact JSON edits (RFC-6902).

Keep the LLM advisory (safe) and let the ACP server synthesize **executable actions** from either:

- deterministic rules (e.g., "set includeDescendants=true for all Drug/Ingredient items that lack it"), or
- structured "action intents" emitted by the model (optional later).

2) Minimal new Action Op for your use case

Add an ACP tool that accepts an **Action Patch** and returns a diff + writes changes if approved.

Action op (schema)

json Copy code

```json
{
  "artifactRef": "path-or-url-to-concept-set.json",
  "ops": [
    {
      "op": "set_include_descendants",
      "where": {
        "domainId": "Drug",
        "conceptClassId": "Ingredient",
        "includeDescendants": false
      },
      "value": true
    }
  ],
  "write": false
}
```

- `where` is a filter over the canonicalized concept-set item shape (see §3).
- `write=false` = dry-run (preview). Toggle to `true` only after user approval in R.

ACP endpoint

`POST /actions/concept_set_edit` → returns:

json Copy code

```json
{
  "plan": "Set includeDescendants=true for Drug/Ingredient entries that la
  "preview_changes": [
    {"conceptId": 789578, "from": {"includeDescendants": false}, "to": {"i
    ...
  ],
  "applied": false,
  "written_to": "same-file-or-new-file-path"
}
```

3) Canonicalization of concept-set items (make patching robust)

ATLAS export layouts vary a bit. Normalize everything server-side to one internal shape:

json

Copy code

```
{
  "items": [
    {
      "concept": {
        "conceptId": 789578,
        "domainId": "Drug",
        "conceptClassId": "Ingredient",
        "...": "..."
      },
      "includeDescendants": false,
      "includeMapped": false,
      "isExcluded": false
    }
  ]
}
```

- During **load**: detect input variant (ATLAS export / list of items / minimal form) and map to this canonical structure.
- Apply actions against the canonical structure.
- During **save**: write back in the **same** structure you loaded (preserve user's format) to avoid surprises.

4) R-side UX (files first)

New R API

```r
# 1) Ask the assistant to propose an executable action (deterministic)
patch <- OHDSIAssistant::proposeIncludeDescendantsPatch(conceptSetRef)

# 2) Preview in R (shows table/diff)
OHDSIAssistant::previewConceptSetPatch(conceptSetRef, patch)

# 3) Apply after approval (writes to file; creates .bak)
OHDSIAssistant::applyConceptSetPatch(conceptSetRef, patch, backup = TRUE)
```

Under the hood

- `proposeIncludeDescendantsPatch()` calls ACP deterministic rule:
  - `POST /actions/concept_set_edit` with `write=false` and `op=set_include_descendants`.
- `previewConceptSetPatch()` pretty-prints `preview_changes` and a unified diff.
- `applyConceptSetPatch()` resubmits the same payload with `write=true` and saves:

    - overwrite original **or** write `*-assistant-v1.json` (configurable).
    - log an entry under `inst/assistant/` with `plan` , `op` , changed IDs, and a hash of pre/post.

Optional: git workflow hook

If `.git/` exists, run:

- `git add <file> → git commit -m "assistant: set includeDescendants for N items (Drug/Ingredient)"`

5) How the LLM participates (but can't break edits)

- Keep your current "notes/patches" output for transparency.
- Extend your system prompt (later) to optionally emit **action intents**:

```json
"actions": [
  {
    "type": "set_include_descendants",
    "where": { "domainId": "Drug", "conceptClassId": "Ingredient", "inc
    "value": true
  }
]
```

- The ACP server **validates** the action and converts it to the same internal op (never trusts free-form JSON Patch paths from the model).
- If the LLM doesn't emit actions, the server can still synthesize them from deterministic rule packs.

6) Open-WebUI wiring

- Keep your workspace the same; **add one tool** exposed to the model (optional):
  - `concept_set_edit(where, value)` → returns a proposed action preview; the UI can display diffs.
- For your current R-driven flow, you don't need the model to call tools; you can call `/actions/concept_set_edit` directly from R after the user clicks "Apply".

7) Safety & validation (non-negotiable)

- **No PHI**: only file-based artifact edits.
- Schema validation before/after write:
  - Ensure every edited item remains a valid concept-set item.
  - Ensure no concept IDs are dropped unintentionally.
- Write policy:
  - Default to **fork** (`*-assistant-v{n}.json`) unless the user sets `overwrite=TRUE` .
  - Always create a `.bak` on first overwrite.
- Deterministic rule visibility:
  - Include in the advice log the exact filter used: `(domainId=='Drug' AND conceptClassId=='Ingredient' AND includeDescendants==false)` and the count of affected items.

8) Near-term ATLAS/WebAPI path

Add a sibling ACP endpoint that targets WebAPI instead of files:

`POST /webapi/concept_set/actions` with:

Copy code

json

```json
{
  "webapi": { "baseUrl": "...", "token": "..." },
  "conceptSetId": 123,
  "ops": [
    {
      "op": "set_include_descendants",
      "where": { "domainId": "Drug", "conceptClassId": "Ingredient", "incl
      "value": true
    }
  ],
  "write": false
}
```

Server flow:

1. Read concept set via WebAPI GET (`/conceptset/{id}` and `/conceptset/{id}/expression`).
2. Canonicalize → apply action → validate.
3. If `write=false` → return preview.
4. If `write=true` :
    - **Create a new concept set** (non-destructive fork): `name = "<old>-assistant-v1"` , POST `/conceptset` and POST expression.
    - Return new IDs, and a link the ATLAS UI can open.

R-side:

r

Copy code

```
OHDSIAssistant::applyConceptSetPatchWebAPI(
  baseUrl, token, conceptSetId, patch, fork = TRUE
)
```

9) Concrete ACP changes to implement now

A) New route & action executor

- `POST /actions/concept_set_edit`
  - Load → canonicalize → filter → mutate → validate → (write?).
  - Return plan, counts, preview rows (conceptId, includeDescendants before/after), file path written (if any).

B) Deterministic rule pack (first rule)

python

Copy code

```python
def op_set_include_descendants(cs_canon, where, value):
    changed = []
    for item in cs_canon["items"]:
        c = item.get("concept", {})
        if (
          (where.get("domainId") is None or c.get("domainId") == where["do
          (where.get("conceptClassId") is None or c.get("conceptClassId")
          (where.get("includeDescendants") is None or item.get("includeDes
        ):
            if item.get("includeDescendants", False) != bool(value):
                changed.append({"conceptId": c.get("conceptId"), "from": i
                item["includeDescendants"] = bool(value)
    return cs_canon, changed
```

C) Validation

- Ensure every changed item still has `concept.conceptId` and booleans for `includeDescendants/includeMapped/isExcluded`.
- Re-expand to original structure when writing back.

10) Minimal R shims to add

```r
#' Propose a deterministic includeDescendants patch
proposeIncludeDescendantsPatch <- function(conceptSetRef) {
  list(
    artifactRef = conceptSetRef,
    ops = list(list(
      op = "set_include_descendants",
      where = list(domainId="Drug", conceptClassId="Ingredient", includeDe
      value = TRUE
    )),
    write = FALSE
  )
}

#' Preview concept-set patch via ACP
previewConceptSetPatch <- function(conceptSetRef, patch) {
  body <- patch; body$artifactRef <- conceptSetRef; body$write <- FALSE
  .acp_post("/actions/concept_set_edit", body)
}

#' Apply concept-set patch via ACP
applyConceptSetPatch <- function(conceptSetRef, patch, backup=TRUE, overwr
  body <- patch; body$artifactRef <- conceptSetRef; body$write <- TRUE
  body$overwrite <- overwrite; body$backup <- backup
  .acp_post("/actions/concept_set_edit", body)
}
```

11) Optional: JSON Patch export (for audit & portability)

When you apply the action, also emit an **RFC-6902 JSON Patch** (paths relative to the canonical structure) into your advice log. This gives you a stable, replayable artifact—even though the actual write is done via the deterministic action op.

12) Tests you can run immediately

- **Dry-run** on your PD Meds concept set: preview shows N changes; no file written.
- **Apply** with `backup=TRUE` : file written, `.bak` created, advice log includes counts + conceptIds.
- **Idempotence**: applying again yields zero changes.
- **Non-Drug** concepts remain untouched.
- **ATLAS** (when you wire it): forked concept set appears with the new name and correct flags.

If you want, I can generate the exact Flask route, the canonicalizer, and the three R functions in ready-to-drop files—plus a tiny unit test that loads `demo/concept_set.json` , runs preview/apply, and asserts the toggles were set only for Drug/Ingredient entries.

proceed

Awesome—here are the **drop-in files** to add executable editing of concept-set JSONs (“set `includeDescendants: true` for Drug/Ingredient entries”) with **preview** and **apply** from R. I’ve included:

- a full replacement for `acp/server.py` (adds `/actions/concept_set_edit` + canonicalizer + writer)
- new R functions (`proposeIncludeDescendantsPatch()` , `previewConceptSetPatch()` , `applyConceptSetPatch()`)
- NAMESPACE update
- a tiny demo/unit script you can run now

---

1) `acp/server.py` (replace the whole file)

python

Copy code

```python
# acp/server.py
import os, json, subprocess, sys, copy, hashlib
from typing import Tuple, List, Dict, Any
import requests
from flask import Flask, request, jsonify
```

```
app = Flask(__name__)

# ---------------------------
# Utilities
# ---------------------------

def load_json(ref: str) -> Any:
    if ref.startswith("http://") or ref.startswith("https://"):
        r = requests.get(ref, timeout=30)
        r.raise_for_status()
        return r.json()
    with open(ref, "r", encoding="utf-8") as f:
        return json.load(f)

def write_json(ref: str, data: Any, backup: bool = True, overwrite: bool =
    """
    Writes JSON to either the same file (overwrite) or to a new sibling wi
    Returns the path written.
    """
    if ref.startswith("http://") or ref.startswith("https://"):
        raise ValueError("Writing to URLs is not supported in this prototy
    base = ref
    if overwrite:
        if backup and os.path.exists(base) and not os.path.exists(base + "
            with open(base, "rb") as src, open(base + ".bak", "wb") as dst
                dst.write(src.read())
        with open(base, "w", encoding="utf-8") as f:
            json.dump(data, f, indent=2, ensure_ascii=False)
        return base
    # fork non-destructively
    root, ext = os.path.splitext(base)
    i = 1
    while True:
        candidate = f"{root}-assistant-v{i}{ext}"
        if not os.path.exists(candidate):
            with open(candidate, "w", encoding="utf-8") as f:
                json.dump(data, f, indent=2, ensure_ascii=False)
            return candidate
        i += 1

def maybe_call_gemini(prompt: str):
    if os.getenv("GEMINI_ACP_ENABLED") != "1":
        return None
    model = os.getenv("GEMINI_MODEL", "gemini-2.0-flash-exp")
    try:
        p = subprocess.run(
            ["gemini", "chat", "--model", model, "--input", prompt],
            capture_output=True, check=True, text=True
        )
        txt = p.stdout.strip()
        start = txt.find("{"); end = txt.rfind("}")
        if start != -1 and end != -1 and end > start:
            return json.loads(txt[start:end+1])
    except Exception as e:
        print(f"[gemini-warning] {e}", file=sys.stderr)
    return None

# --------------------------
# Canonicalization helpers
# --------------------------
```

```
def _to_canonical_concept_set(raw: Any) -> Tuple[Dict[str, Any], Dict[str,
    """
    Convert various concept-set shapes to a canonical structure:
    {
      "items": [
        {
          "concept": { "conceptId": int, "domainId": str, "conceptClassId"
          "includeDescendants": bool,
          "includeMapped": bool,
          "isExcluded": bool
        }, ...
      ]
    }
    Also returns a 'format_descriptor' to reconstruct original shape on wr
    """
    fmt = {"type": None}
    canon = {"items": []}

    if isinstance(raw, dict) and "items" in raw:
        # ATLAS-like
        fmt["type"] = "atlas_items_dict"
        for it in raw.get("items", []):
            c = it.get("concept") or {}
            canon["items"].append({
                "concept": {
                    "conceptId": c.get("conceptId") or c.get("CONCEPT_ID")
                    "domainId": c.get("domainId") or c.get("DOMAIN_ID"),
                    "conceptClassId": c.get("conceptClassId") or c.get("CO
                },
                "includeDescendants": bool(it.get("includeDescendants", Fa
                "includeMapped": bool(it.get("includeMapped", False)),
                "isExcluded": bool(it.get("isExcluded", False))
            })
    elif isinstance(raw, list):
        # Simple list of items
        fmt["type"] = "list_items"
        for it in raw:
            c = it.get("concept") or it
            canon["items"].append({
                "concept": {
                    "conceptId": c.get("conceptId") or c.get("CONCEPT_ID")
                    "domainId": c.get("domainId") or c.get("DOMAIN_ID"),
                    "conceptClassId": c.get("conceptClassId") or c.get("CO
                },
                "includeDescendants": bool(it.get("includeDescendants", it
                "includeMapped": bool(it.get("includeMapped", it.get("incl
                "isExcluded": bool(it.get("isExcluded", it.get("is_exclude
            })
    elif isinstance(raw, dict) and "expression" in raw and "items" in raw[
        # Some exports embed expression.items
        fmt["type"] = "atlas_expression_items"
        for it in raw["expression"]["items"]:
            c = it.get("concept") or {}
            canon["items"].append({
                "concept": {
                    "conceptId": c.get("conceptId") or c.get("CONCEPT_ID")
                    "domainId": c.get("domainId") or c.get("DOMAIN_ID"),
                    "conceptClassId": c.get("conceptClassId") or c.get("CO
                },
                "includeDescendants": bool(it.get("includeDescendants", Fa
                "includeMapped": bool(it.get("includeMapped", False)),
                "isExcluded": bool(it.get("isExcluded", False))
            })
    else:
        raise ValueError("Unsupported concept-set shape")

    return canon, fmt

def _from_canonical_concept_set(canon: Dict[str, Any], fmt: Dict[str, Any]
    """
    Convert canonical structure back into the original shape.
    """
    if fmt["type"] == "atlas_items_dict":
        out = copy.deepcopy(original)
        out_items = []
        for it in canon["items"]:
            out_items.append({
                "concept": {
                    "conceptId": it["concept"]["conceptId"],
                    "domainId": it["concept"]["domainId"],
                    "conceptClassId": it["concept"]["conceptClassId"]
                },
                "includeDescendants": bool(it.get("includeDescendants", Fa
                "includeMapped": bool(it.get("includeMapped", False)),
                "isExcluded": bool(it.get("isExcluded", False))
            })
        out["items"] = out_items
        return out
```

```
if fmt["type"] == "atlas_expression_items":
    out = copy.deepcopy(original)
    out_items = []
    for it in canon["items"]:
        out_items.append({
            "concept": {
                "conceptId": it["concept"]["conceptId"],
                "domainId": it["concept"]["domainId"],
                "conceptClassId": it["concept"]["conceptClassId"]
            },
            "includeDescendants": bool(it.get("includeDescendants", Fa
            "includeMapped": bool(it.get("includeMapped", False)),
            "isExcluded": bool(it.get("isExcluded", False))
        })
    out["expression"]["items"] = out_items
    return out

if fmt["type"] == "list_items":
    out_items = []
    for it in canon["items"]:
        out_items.append({
            "concept": {
                "conceptId": it["concept"]["conceptId"],
                "domainId": it["concept"]["domainId"],
                "conceptClassId": it["concept"]["conceptClassId"]
            },
            "includeDescendants": bool(it.get("includeDescendants", Fa
            "includeMapped": bool(it.get("includeMapped", False)),
            "isExcluded": bool(it.get("isExcluded", False))
        })
```

```python
        return out_items

    raise ValueError("Unsupported format descriptor")

# ---------------------------
# Deterministic rule: set_include_descendants
# ---------------------------

def op_set_include_descendants(canon: Dict[str, Any], where: Dict[str, Any
    changed = []
    for it in canon["items"]:
        c = it.get("concept", {})
        # Filters (None means ignore)
        if where.get("domainId") is not None and c.get("domainId") != wher
            continue
        if where.get("conceptClassId") is not None and c.get("conceptClass
            continue
        if where.get("includeDescendants") is not None and bool(it.get("in
            continue
        cur = bool(it.get("includeDescendants", False))
        if cur != bool(value):
            changed.append({
                "conceptId": c.get("conceptId"),
                "from": cur,
                "to": bool(value)
            })
            it["includeDescendants"] = bool(value)
    return canon, changed

# ---------------------------
# Health & existing endpoints
# ---------------------------

@app.get("/health")
def health():
    return jsonify({"status":"ok"})

@app.post("/tools/propose_concept_set_diff")
def propose_concept_set_diff():
    body = request.get_json(force=True)
    ref = body.get("conceptSetRef")
    study_intent = body.get("studyIntent","")
    cs_raw = load_json(ref)
    # canonicalize preview (no change here)
    items_preview = []
    try:
        canon, _ = _to_canonical_concept_set(cs_raw)
        items_preview = [{"conceptId": i["concept"]["conceptId"], "domainI
    except Exception:
        pass

    findings = []
    patches = []
    risk_notes = []
    plan = f"Review concept set for gaps and inconsistencies given the stu

    # Simple duplicate & mixed-domain checks
    concept_ids = [x["conceptId"] for x in items_preview if x.get("concept
    dups = sorted(set([cid for cid in concept_ids if concept_ids.count(cid
    if len(items_preview) == 0:
        findings.append({"id":"empty_concept_set","severity":"high","impac
    if dups:
        findings.append({"id":"duplicate_concepts","severity":"medium","im
        patches.append({"artifact": ref, "type":"jsonpatch", "ops":[{"op":

    domains = sorted(set([x.get("domainId") for x in items_preview if x.ge
    if len(domains) > 1:
        findings.append({"id":"mixed_domains","severity":"low","impact":"p

    # Optional LLM merge
    prompt = f"""You are checking an OHDSI concept set for study design is
Study intent: {study_intent}
First 20 items: {json.dumps(items_preview)}
Return JSON with fields: findings[], patches[]. Keep patches high-level no
"""
    llm = maybe_call_gemini(prompt)
    if llm:
        for f in llm.get("findings", []):
            if f not in findings: findings.append(f)
        for p in llm.get("patches", []):
            if p not in patches: patches.append(p)

    return jsonify({"plan": plan, "findings": findings, "patches": patches

@app.post("/tools/cohort_lint")
def cohort_lint():
    body = request.get_json(force=True)
    ref = body.get("cohortRef")
    cohort = load_json(ref)
    plan = "Review cohort JSON for general design issues (washout/time-at-
    findings, patches, risk_notes = [], [], []

    pc = cohort.get("PrimaryCriteria", {}) if isinstance(cohort, dict) els
    washout = pc.get("ObservationWindow", {})

    if not washout or washout.get("PriorDays") in (None, 0):
        findings.append({"id":"missing_washout","severity":"medium","impac
        patches.append({"artifact":ref,"type":"jsonpatch","ops":[{"op":"no

    try:
        irules = cohort.get("InclusionRules", [])
        for i, r in enumerate(irules):
            w = r.get("window", {})
            if w and w.get("start",0) > w.get("end",0):
                findings.append({"id":f"inverted_window_{i}","severity":"h
    except Exception:
        pass

    prompt = f"""You are reviewing an OHDSI cohort JSON for general design
Return JSON fields findings[], patches[] (patches are notes, not executabl
Cohort excerpt keys: {list(cohort.keys())[:6]}
"""
    llm = maybe_call_gemini(prompt)
    if llm:
        for f in llm.get("findings", []):
            if f not in findings: findings.append(f)
        for p in llm.get("patches", []):
            if p not in patches: patches.append(p)
```

```python
    return jsonify({"plan": plan, "findings": findings, "patches": patches

# ---------------------------
# NEW: Executable action route
# ---------------------------

@app.post("/actions/concept_set_edit")
def concept_set_edit():
    """
    Body:
    {
      "artifactRef": "path-or-url",
      "ops": [
        {
          "op": "set_include_descendants",
          "where": { "domainId":"Drug","conceptClassId":"Ingredient","incl
          "value": true
        }
      ],
      "write": false,
      "overwrite": false,
      "backup": true
    }
    """
    body = request.get_json(force=True)
    ref = body.get("artifactRef")
    write = bool(body.get("write", False))
    overwrite = bool(body.get("overwrite", False))
    backup = bool(body.get("backup", True))
    ops = body.get("ops", [])
```

```
raw = load_json(ref)
canon, fmt = _to_canonical_concept_set(raw)
before_hash = hashlib.sha256(json.dumps(canon, sort_keys=True).encode(

preview_changes = []
plan_notes = []

# apply ops to canonical
for op in ops:
    if op.get("op") == "set_include_descendants":
        where = op.get("where", {})
        value = bool(op.get("value", True))
        canon, changed = op_set_include_descendants(canon, where, valu
        preview_changes.extend(changed)
        plan_notes.append(f"set_include_descendants where={where} valu
    else:
        return jsonify({"error": f"Unsupported op: {op.get('op')}"}), 

# Prepare response (preview)
after = copy.deepcopy(canon)
after_hash = hashlib.sha256(json.dumps(after, sort_keys=True).encode("
plan = " ; ".join(plan_notes) if plan_notes else "No changes."

written_to = None
applied = False
if write:
    # reconstruct in original format and write
    out_obj = _from_canonical_concept_set(after, fmt, raw)
    written_to = write_json(ref, out_obj, backup=backup, overwrite=ove
        applied = True

    return jsonify({
        "plan": f"Concept set edit: {plan}",
        "preview_changes": preview_changes,   # list of {conceptId, from,
        "counts": {"changed": len(preview_changes)},
        "artifact": ref,
        "applied": applied,
        "written_to": written_to,
        "checksum": {"before": before_hash, "after": after_hash}
    })

if __name__ == "__main__":
    port = int(os.getenv("ACP_PORT", "7777"))
    app.run(host="127.0.0.1", port=port)
```

---

2) R package additions

2.1 `R/OHDSIAssistant/R/ops_concept_set.R` (new)

r

Copy code

```r
#' Build an action patch to set includeDescendants=TRUE for Drug/Ingredien
#' @param conceptSetRef path or URL to concept set JSON
#' @return list payload suitable for /actions/concept_set_edit
proposeIncludeDescendantsPatch <- function(conceptSetRef) {
  list(
    artifactRef = conceptSetRef,
    ops = list(list(
      op = "set_include_descendants",
      where = list(domainId = "Drug", conceptClassId = "Ingredient", inclu
      value = TRUE
    )),
    write = FALSE,
    overwrite = FALSE,
    backup = TRUE
  )
}

#' Preview the concept-set patch (no write)
#' @param conceptSetRef path to concept set JSON
#' @param patch list from proposeIncludeDescendantsPatch()
#' @return server response with preview_changes
previewConceptSetPatch <- function(conceptSetRef, patch) {
  body <- patch
  body$artifactRef <- conceptSetRef
  body$write <- FALSE
  .acp_post("/actions/concept_set_edit", body)
}

#' Apply the concept-set patch (writes a fork or overwrites)
#' @param conceptSetRef path to concept set JSON
#' @param patch list from proposeIncludeDescendantsPatch()
#' @param backup logical; create .bak if overwriting
#' @param overwrite logical; if FALSE, writes -assistant-v{n}.json
#' @return server response with written_to path
applyConceptSetPatch <- function(conceptSetRef, patch, backup = TRUE, over
  body <- patch
  body$artifactRef <- conceptSetRef
  body$write <- TRUE
  body$overwrite <- isTRUE(overwrite)
  body$backup <- isTRUE(backup)
  .acp_post("/actions/concept_set_edit", body)
}
```

2.2 Update `R/OHDSIAssistant/NAMESPACE`

Add exports:

```r
export(proposeIncludeDescendantsPatch)
export(previewConceptSetPatch)
export(applyConceptSetPatch)
```

(Keep your existing exports — just append these three.)

---

3) Demo / test script

Create `scripts/test_concept_set_patch.R`:

```r
# scripts/test_concept_set_patch.R
# Usage:
#   source("scripts/test_concept_set_patch.R")
```

```r
devtools::load_all("R/OHDSIAssistant")
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")

cs_path <- "demo/concept_set.json"  # has Drug+Ingredient items, includeDes

patch <- OHDSIAssistant::proposeIncludeDescendantsPatch(cs_path)

cat("== Preview ==\n")
prev <- OHDSIAssistant::previewConceptSetPatch(cs_path, patch)
print(prev$plan)
print(prev$counts)
head(prev$preview_changes)

stopifnot(prev$counts$changed >= 1)

cat("\n== Apply (fork) ==\n")
applied <- OHDSIAssistant::applyConceptSetPatch(cs_path, patch, backup = T
print(applied$written_to)
stopifnot(isTRUE(applied$applied))

# Idempotence check on the newly written file
cat("\n== Preview on forked file (expect zero changes) ==\n")
prev2 <- OHDSIAssistant::previewConceptSetPatch(applied$written_to, patch)
print(prev2$counts)
stopifnot(prev2$counts$changed == 0)

cat("\nOK: includeDescendants patch applied & idempotent.\n")
```

---

4) Quick runbook

1. **Restart** the ACP bridge:

```bash
./scripts/start_acp.sh
```

2. **Preview & apply** in R:

```r
source("scripts/test_concept_set_patch.R")
```

You should see a preview with N changes, then a new file like `demo/concept_set-assistant-v1.json`, and a second preview reporting `changed == 0`.

---

5) Near-term ATLAS/WebAPI hook (sketch)

- Add `POST /webapi/concept_set/actions` mirroring `/actions/concept_set_edit` but:
    - resolves `conceptSetId` via WebAPI GETs,
    - applies the op on the canonicalized expression,
    - **forks** by POSTing a new concept set (name suffix `-assistant-v{n}`) and its expression back via WebAPI,
    - returns the new `conceptSetId` and optional ATLAS link.

R shim to call it next:

```r
```

```
applyConceptSetPatchWebAPI <- function(baseUrl, token, conceptSetId, overw
  body <- list(
    webapi = list(baseUrl = baseUrl, token = token),
    conceptSetId = conceptSetId,
    ops = list(list(
      op = "set_include_descendants",
      where = list(domainId = "Drug", conceptClassId = "Ingredient", inclu
      value = TRUE
    )),
    write = TRUE,
    fork = isTRUE(fork)
  )
  .acp_post("/webapi/concept_set/actions", body)
}
```

This gives you **actionable edits with approval** for file-based artifacts today, plus a clean path to ATLAS/WebAPI writes (non-destructive forks) next. If you want, I can also add a `--dry-run` CLI to the Flask app for ops-in-terminal and a tiny JSON-Schema validator to guard writes.

I have a deterministic step implemented. I would like to add an example for step 5 above "How the LLM participates (but can’t break edits)." I am not sure of all of the changes that will be needed but I made an attempt at changes to the system prompt where it provides an example of a Patch object. Here I am imagining added an `actions` list. Help me to improve the system prompt that I will pass to the Gemma 3 model, and also a prompt (written completely in markdown) that I will give to the codex programming agent to implement the change.

```
json
### Patch object
A jsonpatch with an advisory `note` op.

#### A) Advisory JSON Patch (notes only)
{
  "artifact": "string echoing the reference
  "type": "jsonpatch",
  "ops": [
    { "op": "note", "path": "/path/within/a
  ]
}

#### B) Advisory with JSON Patch as an acti
6902) applied to the OHDSI artifact (concep
{
  "artifact": "string echoing the reference
  "type": "jsonpatch",
  "ops": [
    { "op": "note", "path": "/path/within/a
  ],
  "actions": [
  {
    "type": "<action identification string>
    "where": { "type": "the type of the OHD
```

```
               "domainId": "artifact domain
               ...,
    "value": <change in value>
  }
 ]
}
```

Great—let's wire in **LLM-emitted** `actions` without letting the model touch bytes directly.

Below are two copy-paste blocks:

1. a **System Prompt** for Gemma 3 (Open-WebUI) that adds an `actions` array to the contract, tightly constrained to the deterministic ops you'll actually execute (today: `set_include_descendants` for concept sets; you can add more later);
2. a **Codex implementation prompt** (pure markdown) that tells your coding agent exactly how to modify the repo: plumb `actions` through the tool responses, add a safe executor endpoint, and expose a tiny R helper to apply the LLM's proposed actions after you approve them.

---

1) System prompt for Gemma 3 (paste into Open-WebUI)

markdown

Copy code

```markdown
You are the **OHDSI Assistant (ACP Model)**. You **must output only a sing

### PURPOSE
```

~~~
Given a short prompt that includes either a **concept set** excerpt or a *
- a brief `plan`,
- structured `findings`,
- advisory `patches` (note-only JSON Patch),
- optional `actions` (strictly-typed, high-level intents the ACP server ca
- `risk_notes` (short caveats).

### STRICT TOP-LEVEL KEYS
Return exactly these keys:
- `plan` (string, ≤300 chars)
- `findings` (array of Finding, may be empty)
- `patches` (array of Patch, may be empty)
- `actions` (array of Action, may be empty)
- `risk_notes` (array of strings, may be empty)

### Finding
```json
{
  "id": "snake_case_identifier",
  "severity": "low|medium|high",
  "impact": "design|validity|portability|performance",
  "message": "Concise human-readable finding (≤220 chars).",
  "evidence": [
    { "ref": "string such as 'conceptId:123' or 'path:/PrimaryCriteria/Obs
  ]
}
~~~

Patch (advisory only; NOT executable edits)

Use JSON Patch **with** `op: "note"` **only** to explain suggested changes.

json

Copy code

```json
{
  "artifact": "string echoing the referenced artifact if provided",
  "type": "jsonpatch",
  "ops": [
    { "op": "note", "path": "/path/within/artifact", "value": { "summary":
  ]
}
```

Action (LLM intent → server-side deterministic edit)

The ACP server will **validate and execute** supported action types. **Do not invent IDs or paths.** Filter only on attributes visible in the excerpt (e.g., `domainId` , `conceptClassId` , booleans).

Supported `type` values (for now):

- `"set_include_descendants"` — for **concept set** items.

Schema:

json

Copy code

```json
{
  "type": "set_include_descendants",
  "where": {
    "domainId": "Drug|Condition|Measurement|Observation|Procedure|Device|<
    "conceptClassId": "e.g., 'Ingredient' if present",
    "includeDescendants": true|false
  },
  "value": true|false,
  "rationale": "≤160 chars",
  "confidence": 0.0
}
```

Rules:

- **Concept sets**: if you detect only ingredient-level **Drug** concepts without `includeDescendants:true` and study intent suggests drug exposure criteria, propose:

  json

  Copy code

  ```json
  {
    "type": "set_include_descendants",
    "where": { "domainId": "Drug", "conceptClassId": "Ingredient", "incl
    "value": true,
    "rationale": "Drug exposures are usually recorded as clinical drug c
    "confidence": 0.8
  }
  ```

- **Cohort definitions**: do **not** emit actions unless you see a specifically supported type for cohorts (if none given, leave `actions: []` and explain in `risk_notes`).

SCOPE & SAFETY

- Never output code fences or commentary; only the JSON object.
- Never fabricate internal database IDs or exact JSON pointer paths.
- If the excerpt is insufficient to target a safe action, omit the action and add a short `risk_notes` message.

- Keep total output **< 15 KB**. No trailing commas.

Example (illustrative; you must return only the JSON object, not this comment)

json Copy code

```json
{
  "plan": "Review drug concept set for descendant coverage and obvious str
  "findings": [
    {
      "id": "suggest_descendants_concept_set",
      "severity": "medium",
      "impact": "design",
      "message": "Ingredient-level Drug concepts lack includeDescendants; 
      "evidence": [{ "ref": "conceptId:789578", "note": "Ingredient in Dru
    }
  ],
  "patches": [
    {
      "artifact": "concept_set_ref",
      "type": "jsonpatch",
      "ops": [
        { "op": "note", "path": "/items", "value": { "summary": "Include d
      ]
    }
  ],
  "actions": [
    {
      "type": "set_include_descendants",
      "where": { "domainId": "Drug", "conceptClassId": "Ingredient", "incl
      "value": true,
      "rationale": "Drug exposures often recorded at clinical drug granula
      "confidence": 0.8
    }
  ],
  "risk_notes": []
}
```

yaml

Copy code

~~~yaml
---

## 2) Codex implementation prompt (give this to your coding agent)

```markdown
# TASK: Add LLM `actions` support (safe, deterministic) to the OHDSI Assis

**Repo context**: The repo has an ACP Flask server at `acp/server.py` and
- `POST /tools/propose_concept_set_diff`
- `POST /tools/cohort_lint`

We need to:
1) Parse any `actions` array emitted by the LLM in the tool routes and inc
2) Add a new executor route that **safely executes** a supplied `actions`
3) Add an R helper to apply LLM actions to a file-based concept set (previ
4) Include a tiny demo script.

## Design rules
- The LLM never supplies raw JSON Pointer paths; it supplies **typed actio
~~~

~~~markdown
- Executor must validate `type`, coerce `value` to bool, and ignore unsupp
- For now, **only concept-set actions** are supported; cohort actions shou

---

## 1) Modify `acp/server.py`

### 1.a Include LLM `actions` in tool responses

In both tool handlers `propose_concept_set_diff` and `cohort_lint`:

- After `llm = maybe_call_gemini(prompt)` and merging `findings/patches`,

```python
actions = []
if llm and isinstance(llm.get("actions", None), list):
    # shallow copy of model-proposed actions (do not execute here)
    actions = llm["actions"]
~~~

- Add `actions` to the returned JSON object:

python Copy code

```python
return jsonify({
    "plan": plan,
    "findings": findings,
    "patches": patches,
    "actions": actions,
    "risk_notes": risk_notes
})
```

Do this in **both** `/tools/propose_concept_set_diff` and `/tools/cohort_lint` (the latter may return `actions: []`).

1.b Add a new executor route: `POST /actions/execute_llm`

Create a new Flask route that accepts:

```json
{
  "artifactRef": "path-or-url-to-concept-set.json",
  "actions": [ /* LLM-proposed actions */ ],
  "write": false,
  "overwrite": false,
  "backup": true
}
```

Behavior:

- Load and canonicalize the concept set (reuse `_to_canonical_concept_set`).
- Iterate actions:
  - Only support `"set_include_descendants"`.
  - Validate `where` keys (`domainId`, `conceptClassId`, `includeDescendants`), coerce `value` to bool.
  - Apply via `op_set_include_descendants(canon, where, value)`.
  - Track `applied_count` and collect `preview_changes` (conceptId/from/to).
  - If unsupported `type`, append an entry to `ignored_actions` with reason.
- If `write==true`, reconstruct original shape with `_from_canonical_concept_set(...)` and write via `write_json(...)` (respect `overwrite` + `backup`).

- Return:

json

Copy code

```json
{
  "plan": "Execute LLM actions (N applied, M ignored).",
  "preview_changes": [ { "conceptId": ..., "from": false, "to": true } ],
  "counts": { "applied": N, "changed": K, "ignored": M },
  "ignored": [ { "type": "foo", "reason": "unsupported" } ],
  "artifact": "<input ref>",
  "applied": true|false,
  "written_to": "<path or null>"
}
```

Code to add (append near other routes):

python

Copy code

```python
@app.post("/actions/execute_llm")
def execute_llm_actions():
    body = request.get_json(force=True)
    ref = body.get("artifactRef")
    actions = body.get("actions", [])
    write = bool(body.get("write", False))
    overwrite = bool(body.get("overwrite", False))
    backup = bool(body.get("backup", True))

    if not isinstance(actions, list):
        return jsonify({"error": "actions must be a list"}), 400
```

```
raw = load_json(ref)
canon, fmt = _to_canonical_concept_set(raw)

total_applied = 0
preview_changes = []
ignored = []

for act in actions:
    atype = act.get("type")
    if atype == "set_include_descendants":
        where = act.get("where", {})
        value = bool(act.get("value", True))
        # sanitize where
        allowed_keys = {"domainId", "conceptClassId", "includeDescenda
        where = {k: v for k, v in where.items() if k in allowed_keys}
        before_len = len(preview_changes)
        canon, changed = op_set_include_descendants(canon, where, valu
        preview_changes.extend(changed)
        # consider action applied if it targeted at least one item
        if len(preview_changes) > before_len:
            total_applied += 1
        else:
            ignored.append({"type": atype, "reason": "no items matched
    else:
        ignored.append({"type": atype, "reason": "unsupported action t

written_to = None
applied = False
if write:
    out_obj = _from_canonical_concept_set(canon, fmt, raw)
```

```
        written_to = write_json(ref, out_obj, backup=backup, overwrite=ove
        applied = True

    return jsonify({
        "plan": f"Execute LLM actions ({total_applied} applied, {len(ignor
        "preview_changes": preview_changes,
        "counts": {"applied": total_applied, "changed": len(preview_change
        "ignored": ignored,
        "artifact": ref,
        "applied": applied,
        "written_to": written_to
    })
```

Note: This executor currently assumes concept-set artifacts. If `raw` doesn't match a concept-set shape, return `400` with a message like `"only concept-set actions are supported in this prototype"`.

---

2) R helper to apply LLM actions

Create `R/OHDSIAssistant/R/ops_llm_actions.R`:

r

Copy code

```r
#' Apply LLM-proposed actions to a concept-set artifact via ACP
#' @param conceptSetRef path to local concept set JSON
#' @param actions list of action objects as returned by the tool response
#' @param preview if TRUE, no write (dry-run)
#' @param overwrite if TRUE, overwrite file; otherwise write -assistant-v{
```

```
#' @param backup if TRUE and overwrite=TRUE, create .bak once
#' @return server response with preview/applied info
applyLLMActionsConceptSet <- function(conceptSetRef, actions, preview = TR
  body <- list(
    artifactRef = conceptSetRef,
    actions = actions %||% list(),
    write = !isTRUE(preview),
    overwrite = isTRUE(overwrite),
    backup = isTRUE(backup)
  )
  .acp_post("/actions/execute_llm", body)
}
```

Update `R/OHDSIAssistant/NAMESPACE` to export:

r

Copy code

```
export(applyLLMActionsConceptSet)
```

---

3) Demo script

Create `scripts/test_llm_actions.R`:

r

Copy code

```
# scripts/test_llm_actions.R
# Assumes you've run lintStudyDesign() and captured the tool response as `
# For demo, we call the concept-sets tool directly to fetch actions.
```

```
devtools::load_all("R/OHDSIAssistant")
OHDSIAssistant::acp_connect("http://127.0.0.1:7777")

concept_set_ref <- "demo/concept_set.json"

# Ask tool to propose actions (via LLM); you can also reuse previous resul
resp <- OHDSIAssistant:::`.acp_post`("/tools/propose_concept_set_diff", li
  conceptSetRef = concept_set_ref,
  studyIntent   = paste(readLines("demo/protocol.md", warn = FALSE), colla
))

actions <- resp$actions
print(actions)

# Preview execution of LLM actions
prev <- OHDSIAssistant::applyLLMActionsConceptSet(concept_set_ref, actions
print(prev$counts)
head(prev$preview_changes)

# Apply after approval (fork file)
applied <- OHDSIAssistant::applyLLMActionsConceptSet(concept_set_ref, acti
print(applied$written_to)
```

---

4) Acceptance criteria

- Tool routes return `actions` (array) when the LLM includes them; otherwise `actions: []`.

- `POST /actions/execute_llm` previews and applies only supported actions; others are reported in `ignored`.
- End-to-end demo:
    1. Run ACP server.
    2. Call `/tools/propose_concept_set_diff` with a Drug/Ingredient concept set lacking `includeDescendants`.
    3. Receive `actions` proposing `set_include_descendants`.
    4. Use `applyLLMActionsConceptSet(..., preview=TRUE)` to see `changed > 0`.
    5. Apply with `preview=FALSE`; a `-assistant-v1.json` file appears; a second preview shows `changed == 0`.

Keep all changes self-contained; do not break existing deterministic `/actions/concept_set_edit` workflow.

```yaml

---

If you want me to also extend the system prompt and executor to a **cohort
::contentReference[oaicite:0]{index=0}
```

Let's take a small detour. Open-webui allows for custom tools to be used within a workspace specific to a model. Examples are provided below. I am thinking that some tools will be very helpful as we advance the prototype. Teach me more about what I need to know to be effective at deciding what should be tool on the LLM side vs something that the ACP server/bridge too provides.

```
python
import os
import requests
from datetime import datetime
from pydantic import BaseModel, Field

class Tools:
    def __init__(self):
        pass

    # Add your custom tools using pure Pyth

    def get_user_name_and_email_and_id(self
        """
        Get the user name, Email and ID fro
        """

        # Do not include a descrption for _
        # The session user object will be p

        print(__user__)
        result = ""

        if "name" in __user__:
            result += f"User: {__user__['na
        if "id" in __user__:
            result += f" (ID: {__user__['id
        if "email" in __user__:
            result += f" (Email: {__user__[

        if result == "":
            result = "User: Unknown"

        return result

    def get_current_time(self) -> str:
        """
        Get the current time in a more huma
        """

        now = datetime.now()
        current_time = now.strftime("%I:%M:
        current_date = now.strftime(
            "%A, %B %d, %Y"
        )  # Full weekday, month name, day,

        return f"Current Date and Time = {c

    def calculator(
        self,
        equation: str = Field(
            ..., description="The mathemati
        ),
    ) -> str:
        """
        Calculate the result of an equation
        """
        # Avoid using eval in production co
        # https://nedbatchelder.com/blog/20
        try:
            result = eval(equation)
            return f"{equation} = {result}"
        except Exception as e:
            print(e)
            return "Invalid equation"

    def get_current_weather(
        self,
        city: str = Field(
            "New York, NY", description="Ge
        ),
    ) -> str:
        """
        Get the current weather for a given
        """

        api_key = os.getenv("OPENWEATHER_AP
        if not api_key:
            return (
                "API key is not set in the 
            )

        base_url = "http://api.openweatherm
        params = {
            "q": city,
            "appid": api_key,
            "units": "metric",  # Optional:
        }

        try:
            response = requests.get(base_ur
            response.raise_for_status()  #
            data = response.json()

            if data.get("cod") != 200:
                return f"Error fetching wea

            weather_description = data["wea
            temperature = data["main"]["tem
            humidity = data["main"]["humidi
            wind_speed = data["wind"]["spee

            return f"Weather in {city}: {te
        except requests.RequestException as
            return f"Error fetching weather
```

You’re on the right track. The short version:

- **Let Open-WebUI “tools” be thin, stateless, low-risk helpers the LLM can call to *orchestrate* and *preview*.**
- **Keep anything stateful, write-heavy, schema-sensitive, or secret-bearing in the ACP bridge (server) with explicit approval gates.**

Here’s a practical decision playbook plus ready-to-adapt tool ideas/snippets.

What belongs where?

Put in Open-WebUI Tools (LLM side)

Use these for **pure functions** or **gated orchestration** with predictable IO:

- **Shaping / construction helpers**
  Build *Action* payloads the ACP understands (e.g., `set_include_descendants`), validate required fields, default sane values.
- **Preview-only calls**
  Call ACP **dry-run** endpoints and return diffs (never write).
- **Light summarization / formatting**
  Create compact tables or summaries of previews for the chat, *without* changing artifacts.
- **Capability discovery**
  List supported ACP ops; surface constraints to the LLM so it stays on the rails.
- **Small, deterministic lookups** that don't require secrets (or use per-workspace env vars with least privilege).

Why: Tools invoked by the model should be idempotent, fast, and safe if called multiple times. They keep the LLM grounded without handing it the keys to your data stores.

Keep in ACP (server/bridge)

Anything that needs **authority, audit, or schema guarantees**:

- **Reads/writes of artifacts** (files + ATLAS/WebAPI objects).
- **JSON schema validation** and **canonicalization**.
- **Deterministic edit engines** (convert `actions` → exact JSON edits; generate RFC-6902 if you want).
- **Security & secrets** (WebAPI tokens, DB creds).

- **Heavy ops** (DQD/Achilles fetch, cohort counts, concept vocabulary expansion, async jobs).
- **Audit logs, backups, git commits, idempotency**.

Why: The ACP can enforce approvals, validate inputs, log diffs, and fail safe.

Checklist to decide “Tool vs ACP”

- **Side-effects?** → ACP.
- **Needs secrets?** → ACP.
- **Schema coupling / fragile pointers?** → ACP.
- **May run long or require retries?** → ACP (and maybe async job pattern).
- **Pure transform / preview / shape JSON?** → Open-WebUI tool.
- **LLM benefit from immediate feedback to refine a plan?** → Tool that calls ACP in **preview** mode only.

Suggested first tools (LLM side)

Below are small, targeted tools that keep the model productive and safe.

1) Discover capabilities

Lets the model ask “what actions can I propose?”

python Copy code

```python
from pydantic import BaseModel, Field
import requests, os
```

```
class ListCapabilitiesOut(BaseModel):
    actions: list[str] = Field(..., description="Supported action types")
    version: str

class Tools:
    def list_acp_capabilities(self) -> ListCapabilitiesOut:
        """
        List deterministic action types the ACP can execute.
        """
        # Option A: hardcode now; later call ACP /capabilities
        return ListCapabilitiesOut(actions=["set_include_descendants"], ve
```

2) Build an action payload (no side-effects)

Keeps the LLM from inventing shapes.

python

Copy code

```python
from pydantic import BaseModel, Field

class BuildDescendantsIn(BaseModel):
    domainId: str = Field(..., description="Expected 'Drug' for this actio
    conceptClassId: str = Field("Ingredient")
    includeDescendants: bool = Field(False, description="Match items curre
    value: bool = Field(True, description="Set to True to include descenda

class BuildActionsOut(BaseModel):
    actions: list[dict]
```

```python
class Tools:
    def build_set_include_descendants(self, cfg: BuildDescendantsIn) -> Bu
        """
        Construct a well-formed 'set_include_descendants' actions list for
        """
        act = {
            "type": "set_include_descendants",
            "where": {
                "domainId": cfg.domainId,
                "conceptClassId": cfg.conceptClassId,
                "includeDescendants": cfg.includeDescendants
            },
            "value": cfg.value,
            "rationale": "Capture clinical drug codes under ingredient con
            "confidence": 0.7
        }
        return BuildActionsOut(actions=[act])
```

3) Preview an action (dry-run via ACP)

This is safe; no writes.

python

Copy code

```python
class PreviewIn(BaseModel):
    artifactRef: str = Field(..., description="Path/URL to concept set JSO
    actions: list[dict] = Field(...)

class PreviewOut(BaseModel):
    changed: int
    preview_changes: list[dict]

class Tools:
    def preview_actions(self, inp: PreviewIn) -> PreviewOut:
        """
        Call ACP /actions/execute_llm with write=false to get a diff.
        """
        acp = os.getenv("ACP_BASE_URL", "http://127.0.0.1:7777")
        r = requests.post(f"{acp}/actions/execute_llm", json={
            "artifactRef": inp.artifactRef,
            "actions": inp.actions,
            "write": False
        }, timeout=30)
        r.raise_for_status()
        data = r.json()
        return PreviewOut(changed=data["counts"]["changed"], preview_chang
```

4) (Optional) Ask for approval token

You can add a minimal “approval” tool that just echoes a signed token the ACP checks once—useful if you want an explicit human-in-loop step in chat. (Token generation/checking belongs in ACP, but the *request/return* can be brokered by a tool.)

Patterns to keep you safe

- **Preview–Apply dance**: tools trigger **preview**; only R or ATLAS UI triggers **apply** (or a second, explicit tool call that includes a human “approved=true” flag).

- **Idempotency**: ACP should return 0 changes on a second preview/apply with identical state.
- **Schema guards**: ACP validates before/after; rejects writes if shape drifts.
- **Action whitelisting**: ACP ignores unknown `type` s and reports them (counts.ignored).
- **Backups + forks**: ACP writes `*-assistant-vN.json` by default; overwrites require an explicit flag.
- **Audit trail**: ACP emits a compact record (who/when/what/why) and stores the RFC-6902 diff if you generate it.

Where this goes next (Atlas/WebAPI)

- Add ACP endpoints that **fork** ATLAS objects (concept sets, cohort defs) under a suffix (e.g., `-assistant-v1`) and POST back expressions.
- Expose a **preview tool** in Open-WebUI that calls ACP against WebAPI IDs (read-only).
- Keep the **apply** step outside the LLM (R/Atlas UI click), or require an explicit confirmation tool call that echoes back preimage hashes the ACP returns (prevents TOCTOU errors).

Small usability tips

- Return **machine-first JSON** from tools (no prose), plus a short `human_summary` string if you want the model to read it.
- Put **timeouts** on tool network calls; report concise errors (model can recover).

- Keep tool inputs tiny; pass **refs** (file paths, IDs), let ACP fetch heavy payloads.

If you want, I can package those four tools into a single `tools.py` you can drop into Open-WebUI and wire to your current ACP, along with an updated system prompt that nudges the model to **call tools to preview**, and then **propose actions** only when confident.

That was helpful but I need just a bit more education on how tools are used by the LLM model and how pydantic "wires" the tools so that they are triggered when the model receives a prompt.

Great question. Here's the mental model you need for **how Open-WebUI tools work** and **how Pydantic "wires" them** so a model can call them from plain language.

How tool calling actually works

1. **You register tools** (Python functions/methods) in Open-WebUI.
   - Open-WebUI inspects your class and builds a **tool schema**: name, description, and **argument schema**.
2. **That schema is shown to the model** at inference time.
   - Think "hey model, here are callable tools; if useful, call one by name with these arguments."
   - If your model natively supports function calling (OpenAI-style), WebUI uses that. If it doesn't (e.g., many local LLMs), WebUI **shims** it with a system prompt

+ output parsing so tool calls still work.

3. **The LLM decides to call a tool** (or not).
   - It emits a structured "call" `{name: "...tool...", arguments: {...}}` instead of normal text.
   - WebUI catches that, validates the arguments, runs your Python function, captures the return value, then **feeds the tool result back** to the model as context ("tool result").
4. **The model continues** with that new information.
   - It might call more tools or produce the final JSON/answer.

So: **model** → **choose tool** → **WebUI validates & executes** → **result** → **model continues**.

Where Pydantic fits

Pydantic provides the **argument contract**:

- When you annotate your tool parameters with **type hints** (and/or Pydantic `BaseModel` with `Field(...)`), Open-WebUI can:
  - Build a **JSON Schema** (name, types, required/optional, descriptions).
  - **Validate** incoming tool-call arguments from the model.
  - Give the model **clear affordances** so it fills args correctly (descriptions matter!).

If validation fails, WebUI sends an error back (as a tool error message), and the model can try again with corrected arguments.

Minimal patterns

A. Simple function with Field descriptions

python Copy code

```
from pydantic import Field

class Tools:
    def get_current_weather(
        self,
        city: str = Field(..., description="City name, e.g., 'Pittsburgh, 
        units: str = Field("metric", description="'metric' or 'imperial'")
    ) -> str:
        """
        Get the current weather for a city.
        """
        ...
        return "Weather in Pittsburgh: 4°C"
```

- `Field(...)` → required.
- `Field("metric", ...)` → default.
- The docstring becomes the **tool's description**; Field descriptions become the **param help** the model sees.

B. Strongly-typed request/response via BaseModel

python

Copy code

```
from pydantic import BaseModel, Field

class WeatherIn(BaseModel):
    city: str = Field(..., description="City, e.g., 'Austin, TX'")
    units: str = Field("metric", description="'metric' or 'imperial'")
```

```
class WeatherOut(BaseModel):
    temp: float
    summary: str

class Tools:
    def get_weather(self, inp: WeatherIn) -> WeatherOut:
        """Get current weather with structured output."""
        temp = 21.0; summary = "clear"
        return WeatherOut(temp=temp, summary=summary)
```

- Many tool frameworks (including Open-WebUI) happily accept a **single Pydantic argument**; it creates a single JSON object schema under `inp`.
- Returning a Pydantic model yields a **nice, predictable JSON result** back to the model.

What makes a tool “callable” by the model?

- **Name**: method/function name (keep it short, verb-first).
- **Description**: docstring—this is what the LLM reads to decide to use it. Be explicit about *when* to use the tool.
- **Arguments**: Pydantic/Field types + descriptions. Prefer primitives, enums, and shallow objects.
- **Determinism**: Tools should be idempotent and quick. (For writes, add an explicit `apply=true` flag; previews default.)

Enabling it in Open-WebUI

- Place your `Tools` class in the workspace tools file, exactly like your example.
- Ensure **Function Calling** is enabled in model settings (some local runners label it “tools” or “function_call”).

- For models without native function calling, WebUI auto-prompts to simulate it—your docs/descriptions become even more important.

Best-practice design for your OHDSI prototype

- **LLM-side tools** (safe; no writes):
  - Build action payloads (e.g., `build_set_include_descendants`).
  - Call ACP in **dry-run** (`preview_actions`) and return diffs.
  - List ACP capabilities (`list_acp_capabilities`).
- **ACP server** (stateful; writes/validation/audit):
  - Execute actions; write files/WebAPI artifacts; backups; schema checks.

This keeps the LLM helpful without giving it direct write power.

Debugging & ergonomics

- **Make descriptions imperative and narrow**: “Set includeDescendants on Drug/Ingredient items after preview shows >0 changes.”
- **Validate strictly** with Pydantic—bad args fail fast and teach the model.
- **Echo inputs** on failures (“expected units: 'metric' | 'imperial'”).
- **Log** each tool call with args and duration.
- **Return concise, model-readable JSON**, plus an optional short `human_summary` string if you want a friendlier chat trace.

Tiny end-to-end example for your flow

Tool 1 (build action):

python

Copy code

```
from pydantic import BaseModel, Field

class BuildDescendantsIn(BaseModel):
    domainId: str = Field(..., description="Use 'Drug' for PD meds.")
    conceptClassId: str = Field("Ingredient")
    includeDescendants: bool = Field(False, description="Match items lacki
    value: bool = Field(True, description="Set to True to include descenda

class BuildActionsOut(BaseModel):
    actions: list[dict]
    human_summary: str

class Tools:
    def build_set_include_descendants(self, cfg: BuildDescendantsIn) -> Bu
        act = {
            "type": "set_include_descendants",
            "where": {
                "domainId": cfg.domainId,
                "conceptClassId": cfg.conceptClassId,
                "includeDescendants": cfg.includeDescendants
            },
            "value": cfg.value,
            "rationale": "Capture clinical drug codes under ingredients.",
            "confidence": 0.7
        }
        return BuildActionsOut(
            actions=[act],
            human_summary="Built 1 action to set includeDescendants=true f
        )
```

Tool 2 (preview via ACP; no write):

```python
class PreviewIn(BaseModel):
    artifactRef: str = Field(..., description="Path/URL to concept set JSO
    actions: list[dict] = Field(...)

class PreviewOut(BaseModel):
    changed: int
    preview_changes: list[dict]
    human_summary: str

class Tools:
    def preview_actions(self, inp: PreviewIn) -> PreviewOut:
        import os, requests
        acp = os.getenv("ACP_BASE_URL", "http://127.0.0.1:7777")
        r = requests.post(f"{acp}/actions/execute_llm", json={
            "artifactRef": inp.artifactRef,
            "actions": inp.actions,
            "write": False
        }, timeout=30)
        r.raise_for_status()
        data = r.json()
        return PreviewOut(
            changed=data["counts"]["changed"],
            preview_changes=data["preview_changes"],
            human_summary=f"Preview: {data['counts']['changed']} items wou
        )
```

Flow in chat (from the model's perspective):

- User: "Update the PD meds concept set to include descendants where appropriate."
- Model: calls `build_set_include_descendants` → gets `actions`.
- Model: calls `preview_actions` with `artifactRef` + `actions` → sees `changed: 14`.
- Model: asks the user for approval; once approved, your **R client** (or a separate "apply" tool if you choose) does the write through ACP.

That's the wiring: Pydantic shapes the contract; Open-WebUI exposes it to the LLM; the model chooses to call tools; WebUI validates, executes, and returns; the model continues with better context.

So, it has been a day of development and my code agent and I have made some good progress. So much that we moved on to extend to a new use case. Check it out:

"

would like to add a use case for the ACP server/bridge that leverage the LLM to convert a study research question description into one or more candidate phenotype definitions. Here is the envisioned workflow:

1) The user writes their research question (e.g., see ./AgenticStudyAssistant/demo/protocol.md)

2) The user requests the assistance to identify the electronic health records phenotype definitions (aka "cohort definitions") that would be relevant. If R, they would call a function in the OHDSIAssistant package pointing to the research protocol markdown file.

3) The ACP would communicate the request to the LLM along with a list of identifiers and names for existing OHDSI phenotype definitions from ./AgenticStudyAssistant/demo/Cohorts.csv.

4) The model will reply with JSON object that `phenotype_recommendations` containing a list of zero or more suggested phenotypes from the Cohorts.csv that would potentially be useful for the study. Each recommendation will come with a justification.

5) The ACP server/bridge will pass this information in R along with a prompt the user to review the options and select which ones (if any) that they think would be relevant to their study.

6) The user makes a selection and the OHDSIAssistant then retrieves the cohort definitions by loading the phenotype R library (`library(PhenotypeLibrary)`), getting the phenotype definitions by cohort id (e.g., `cohortDefinitionSet <- PhenotypeLibrary::getPlCohortDefinitionSet( cohortIds = c(33,1197))` a\

nd then writing the JSON cohort definitions to the project folder in the same folder where the protocol.md file is located. You can find the full code for PhenotypeLibrary in a sub-folder called "PhenotypeLibrary".

7) The OHDSIAssistant then asks the user wants a review for suggested improvements to the phenotype definitions based on the study goals and the data source information. If they agree, the ACP server/bridge sends the specific details of the cohort definitions to the LLM prompting for a review based on the study goals in protocol.md, the concept sets, and logic of the definition.

7b) Optionally, the user could have the OHDSIAssistant generate the cohorts on their data source (specified by dbDetails.json) and run cohort characterization to download the reports generated (all deidentified aggregate data) that could be sent to the LLM for further analysis and better suggestions. For an example of code that will accomplish this, see ./AgenticStudyAssistant/scripts/testPhenotypeRun.R and note that the code for CohortDiagnost\
ics in a sub-folder by the same name.

) The LLM replies to the ACP server with `phenotype_improvements` containing a list of zero or more suggested phenotype changes (summaries along with `actions`) to concept sets or logic.

9) The OHDSIAssistant presents the report of suggestions to the user for review. The user can select to have the OHDSIAssistant make the changes.

9b) The OHDSIAssistant can at this point request R coding assistance from the LLM through the ACP server/bridge. The user would have to approve in a prompt trigged by the OHDSIAssistant. After sending the request, response to the ACP server would come back as a `code_suggestion` that the ACP can ask the user if it OK to implement.

This workflow demands a complex set of changes along with improvements to the system prompt of the LLM (see ./AgenticStudyAssistant/agentstudyassistant-system-prompt.md). So, let's work on this in steps.
"

This is a great next step. Here's a focused, build-ready plan to add the **"research-question → candidate phenotypes"** workflow, with tight contracts, server routes, R API, and LLM prompt updates. I'll keep the edits incremental so you can land them quickly.

1) Data contracts (JSON you'll pass around)

1.1 Input to LLM (recommendation stage)

json

Copy code

```json
{
  "task": "phenotype_recommendations",
  "study_intent_markdown_ref": "path-or-url-to/protocol.md",
  "study_intent_excerpt": "short excerpt (<=1500 chars) of protocol.md",
  "candidate_catalog": [
    { "cohortId": 33, "name": "Parkinson's disease (PheValuator)", "tags":
    { "cohortId": 1197, "name": "Parkinsonism broad", "tags": ["Condition"
  ],
  "selection_limit": 10
}
```

1.2 LLM output (recommendation stage)

json

Copy code

```json
{
  "plan": "Rank phenotypes by conceptual fit to research question and like
  "phenotype_recommendations": [
    {
      "cohortId": 33,
      "name": "Parkinson's disease (PheValuator)",
      "justification": "Matches Dx-based case definition; validated; align
      "confidence": 0.82,
      "notes": ["Use ingredient-based exclusion for neuroleptics in exposu
    }
  ],
  "risk_notes": []
}
```

1.3 Input to LLM (improvement stage)

json Copy code

```json
{
  "task": "phenotype_improvements",
  "study_intent_excerpt": "short excerpt",
  "cohort_definitions": [
    {
      "cohortId": 33,
      "name": "Parkinson's disease (PheValuator)",
      "definition": { "...": "ATLAS JSON" },
      "concept_sets": [{ "id": 0, "name": "PD Dx", "expression": { "items"
      "source_characterization": {
        "achilles": { "summary": { "personCount": 123456 } },
        "cohort_diagnostics": { "incidence_rate": 3.2, "age_sex": [/* … */
      }
    }
  ]
}
```

1.4 LLM output (improvement stage)

json Copy code

```json
{
  "plan": "Check washout, inclusion/exclusion alignment with study aims, o
  "phenotype_improvements": [
    {
      "cohortId": 33,
      "summary": "Set includeDescendants for PD meds and exclude antipsych
      "actions": [
        {
          "type": "set_include_descendants",
          "target": "concept_set",
          "where": { "domainId": "Drug", "conceptClassId": "Ingredient", "
          "value": true,
          "rationale": "Exposure often recorded as clinical drug codes."
        }
      ],
      "findings": [
        {
          "id": "missing_washout",
          "severity": "medium",
          "impact": "validity",
          "message": "PriorDays washout < 365; consider >=365 to reduce pr
          "evidence": [{ "ref": "path:/PrimaryCriteria/ObservationWindow",
        }
      ]
    }
  ],
  "risk_notes": []
}
```

2) System-prompt additions (Gemma 3)

Add two **new tasks** to your system prompt; keep the “JSON-only” rule.

New sections to append to your current prompt

- **Task:** `phenotype_recommendations`

  Input provides a study-intent excerpt and a `candidate_catalog` from Cohorts.csv.

  Output: `phenotype_recommendations` (0..N), each with `cohortId`, `name`, `justification` (≤300 chars), optional `notes[]`, and `confidence` in [0,1]. Do not invent cohortIds. If uncertain, return an empty list and add a brief `risk_notes` item.
- **Task:** `phenotype_improvements`

  Input provides `cohort_definitions[]` (ATLAS JSON snippets), `concept_sets[]`, and optional source characterization summaries.

  Output: `phenotype_improvements` (0..N). For each:
  - `summary` (≤300 chars),
  - `findings[]` (same schema you already use),
  - optional `actions[]` (strictly whitelisted types you support server-side; today: `"set_include_descendants"` with the same `where/value` schema you used earlier),
  - `risk_notes[]`.

    Never output direct JSON Patches; only high-level `actions`.

Keep total output under 15 KB and never include code fences.

3) ACP server endpoints to add

3.1 Recommend phenotypes (reads protocol + Cohorts.csv; no side effects)

```css
POST /assist/phenotypes/recommend
Body: {
  "protocolRef": "path-or-url-to/protocol.md",
  "cohortsCsvRef": "path-or-url-to/Cohorts.csv",
  "selection_limit": 10
}
→ { plan, phenotype_recommendations[], risk_notes[] }
```

Server flow:

1. Load `protocol.md`, clip to ~1500 chars, load/parse `Cohorts.csv` → array of `{cohortId, name, tags, source}`.
2. Call LLM with `task=phenotype_recommendations`.
3. Validate that every returned `cohortId` exists in the CSV; drop unknowns; return.

3.2 Improve phenotypes (pre-review; no writes)

```css
POST /assist/phenotypes/improve
Body: {
  "protocolRef": ".../protocol.md",
  "cohortIds": [33,1197],
  "phenotypeLibrary": { "enabled": true, "args": {...} },
  "characterization": { "include": false }  // later: attach Achilles/CD r
}
→ { plan, phenotype_improvements[], risk_notes[] }
```

Server flow:

1. If `phenotypeLibrary.enabled`, call into R (or shell script) to get definitions via `PhenotypeLibrary::getPlCohortDefinitionSet()`; otherwise resolve from local

files.

2. Attach optional light characterization (if enabled).
3. Call LLM with `task=phenotype_improvements`.
4. Return `phenotype_improvements` (actions are **not executed** here).

(You already have the deterministic action executor for concept sets; reuse that when user approves.)

4) R package API

Add three user-facing helpers:

```r
# 1) Ask model to recommend phenotypes from Cohorts.csv
recommendPhenotypes <- function(protocolRef, cohortsCsvRef, limit = 10)

# 2) Given selected cohortIds, pull definitions via PhenotypeLibrary and a
reviewSelectedPhenotypes <- function(protocolRef, cohortIds, usePhenotypeL

# 3) Apply any improvement actions aimed at concept sets (with preview/app
applyPhenotypeActions <- function(projectDir, improvements, preview = TRUE
```

Typical flow in R:

```r
rec <- recommendPhenotypes("demo/protocol.md", "demo/Cohorts.csv", limit =
print(rec$phenotype_recommendations)  # let the user pick

sel <- c(33, 1197)
```

```
imp <- reviewSelectedPhenotypes("demo/protocol.md", sel, usePhenotypeLibra

# Preview actions for concept-set changes on the local JSON copies you wro
applyPhenotypeActions("demo", imp, preview = TRUE)
# After approval:
applyPhenotypeActions("demo", imp, preview = FALSE, overwrite = FALSE, bac
```

Implementation notes:

- `recommendPhenotypes()` → POST `/assist/phenotypes/recommend`.
- `reviewSelectedPhenotypes()` → POST `/assist/phenotypes/improve`; before sending, call the PhenotypeLibrary code to **write the retrieved cohort JSON** right next to `protocol.md` (e.g., `demo/cohorts/<id>.json`) so subsequent action previews can target files.
- `applyPhenotypeActions()` → iterate `phenotype_improvements[].actions[]` and route supported actions to your existing `/actions/execute_llm` or `/actions/concept_set_edit`.

5) Open-WebUI tools (optional, safe-only)

Add thin tools the LLM can call to **preview** and **prepare**:

- `list_catalog_from_csv(cohortsCsvRef)` → returns structured list for the model, deduped and capped.
- `preview_actions(artifactRef, actions[])` → calls your ACP `execute_llm` with `write=false`.
- *(Do not add write tools; keep writes on ACP or in R.)*

6) Guardrails & scoring

- **ID validation**: any `cohortId` from the model must exist in the supplied `Cohorts.csv`.
- **Deterministic action whitelist**: currently only `set_include_descendants` targeting concept sets; ignore the rest with an “unsupported action type” note.
- **Confidence threshold**: in R, present only `confidence >= 0.5` by default (toggleable).
- **Explainability**: always show `justification` and, if available, links from Cohorts.csv to provenance docs.

7) Quick code hooks (names you can drop in)

ACP (Flask):

- `POST /assist/phenotypes/recommend`
  - loads markdown & CSV (use `pandas.read_csv` with `dtype` for `cohortId`), builds prompt, returns `phenotype_recommendations`.
- `POST /assist/phenotypes/improve`
  - resolves definitions (optionally via an Rscript wrapper that emits JSON to stdout), builds prompt with trimmed JSON (first ~1–2 concept sets per cohort if huge), returns `phenotype_improvements`.

R:

- `recommendPhenotypes()` → POST recommend; print a small table with `cohortId`, `name`, `confidence`, `justification`.
- `reviewSelectedPhenotypes()` → fetch/write JSONs (via PhenotypeLibrary), POST improve; save the LLM report under `inst/assistant/phenotype_improvements_<ts>.json`.

- `applyPhenotypeActions()` → for each improvement with actions: call your existing preview/apply endpoints; log under `inst/assistant/` .

8) Example minimal prompt body for recommendation

(Server composes this; model sees only JSON and your system prompt)

```json
{
  "task": "phenotype_recommendations",
  "study_intent_excerpt": "Identify clinical risk factors for Parkinson's 
  "candidate_catalog": [
    {"cohortId":33,"name":"Parkinson's disease (PheValuator)","tags":["Con
    {"cohortId":1197,"name":"Parkinsonism broad","tags":["Condition"],"sou
    {"cohortId":501,"name":"Essential tremor","tags":["Condition"],"source
  ],
  "selection_limit": 10
}
```

9) Testing plan (fast)

1. **Happy path**: Cohorts.csv with 10+ rows; protocol.md present.
    - `/assist/phenotypes/recommend` returns ≥1 recommendation with valid IDs.
2. **Edge**: CSV missing an ID the model suggests → server filters it out; `risk_notes` empty.
3. **Improvement**: selected IDs resolve via PhenotypeLibrary; `/assist/phenotypes/improve` yields at least one action (concept set descendants) and one finding (washout).

4. **Apply**: `applyPhenotypeActions(preview=TRUE)` shows `changed > 0`; applying writes `-assistant-v1.json`; second preview shows `changed == 0`.

10) Near-term Atlas/WebAPI

Mirror the file flow with WebAPI:

- `POST /assist/phenotypes/recommend` stays the same.
- For improvements:
  - Read definitions via WebAPI (GET `/cohortdefinition/{id}` + expression),
  - **Fork** on apply (POST new cohort + POST expression), never in-place,
  - Return `newCohortId` and ATLAS link.

If you want, I can generate the actual Flask route stubs and the three R functions with minimal implementations that call your existing utilities and save reports under `inst/assistant/`.

That was a good summary. I want to dig into the various ways to make the phenotype recommendations more clinically accurate. While we are working only on a prototype, I can foresee that one of the issues is that there can be thousands of phenotype definitions and passing them entirely as a catalog to the model is not a good approach. For example, the JSON structure pasted below is scalable to thousands of items in the catalog but it is inefficient and will likely confuse the LLM. Also, techniques such as embedding the phenotype definition descriptions and the user's study intent statement, and clinically reasonable inferences that the model could make from the statement, could whittle down the many to a few based on approximate similarity matching. Consider options to address this such as the use of MCP by the ACP server and tools given to the LLM and let me know what you thing might work best.

json

```
{
  "task": "phenotype_recommendations",
  "study_intent_markdown_ref": "path-or-url
  "study_intent_excerpt": "short excerpt (<
  "candidate_catalog": [
    { "cohortId": 33, "name": "Parkinson's 
    { "cohortId": 1197, "name": "Parkinsoni
  ],
  "selection_limit": 10
}
```

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

Output from model (unchanged shape):

```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
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 fo
- phenotype_fetch_definition(cohortId, truncate=true): safe, truncated ATL

POLICY:
- Use tools for recall or preview; do not attempt to list all phenotypes fr
- 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.