CaRMS Project Mastery Plan (Updated)

Last update: 2026-02-12 — semantic search foundation shipped.

This guide is designed to help you build **deep, end-to-end ownership** of this project across architecture, data, code, runtime behavior, and product/DS extension ideas.

What changed in this update

- Added pgvector-backed table `gold_program_embedding` via Alembic migration `20260212_0002`.
- New Dagster asset `gold_program_embeddings` that encodes `gold_program_profile.description_text` with `all-MiniLM-L6-v2`.
- New FastAPI endpoint `POST /semantic/query` for semantic search (optional LangChain QA summary when `OPENAI_API_KEY` is set).
- API contract + README refreshed to include the semantic endpoint and new gold table.
- Dependencies updated to include `langchain`, `langchain-community`, and `langchain-openai`.

Workstreams still planned

1) Match simulation / what-if engine (`carms/analytics/simulation.py`, `/analytics/simulate`).

2) Preference modeling (`carms/analytics/preferences.py`, `/analytics/preferences`).

3) AWS deployment path (Terraform for RDS + ECS + ECR + S3; CI plan).

CaRMS Project Mastery Plan (Interview-Focused)

This guide is designed to help you build **deep, end-to-end ownership** of this project across architecture, data, code, runtime behavior, and product/DS extension ideas.

Goal and outcomes

By the end of this plan, you should be able to confidently explain:

1. **Why the platform exists** and which user problems it solves.

2. **How data flows** from raw files into API-ready gold tables.
3. **How one API call is served** from route -> query -> schema -> response.
4. **How quality, performance, and security are enforced** in practice.
5. **What you would improve in your first 30 days** as a junior DS/DE.

Target total time: **6–9 hours**.

Phase A — Conceptual understanding (1–2 hours)

Step A1) Read README.md end-to-end

Focus on: - Platform intent and scope. - Two run modes (UI-only vs full platform). - Architecture diagram and endpoint surface. - Security and rate limiting behaviors.

Output artifact: a 1-page note titled "System in 90 seconds" answering: - What the system does. - Why it uses bronze/silver/gold. - Why Dagster + FastAPI + Postgres were selected.

Step A2) Read docs in this exact order

1. `docs/architecture.md`
2. `docs/data-dictionary.md`
3. `docs/api-contract.md`
4. `docs/performance.md`

As you read, maintain a two-column table: - **Column A: claim** (e.g., "silver standardizes and validates rows"). - **Column B: where it is implemented** (file path / function / model).

Step A3) Write your own 8–10 architecture bullets

Use your own wording. Keep each bullet concise and testable.

Suggested structure for your bullets - 1 bullet: business objective. - 2 bullets: ingestion and transformation. - 2 bullets: serving layer and API. - 1 bullet: security/rate limit. - 1 bullet: orchestration/operations. - 1 bullet: performance strategy. - 1 bullet: roadmap/future capability.

Interview check: if you cannot explain each bullet without notes, revisit docs.

Phase B — Code tracing (2–3 hours)

Step B1) Start from `carms/pipelines/definitions.py`

Understand the asset/check registration and job wiring: - Which asset groups are loaded. - What checks are attached. - How materialization order is implied

through dependencies.

Step B2) Trace one record bronze -> silver -> gold

Pick one `program_stream_id` and follow it through: - Bronze models/assets (`carms/pipelines/bronze`, `carms/models/bronze.py`). - Silver transforms (`carms/pipelines/silver`, `carms/models/silver.py`). - Gold curation (`carms/pipelines/gold`, `carms/models/gold.py`).

Document for each stage: - Input fields. - Transformation/validation logic. - Output fields. - Potential failure or null/edge conditions.

Step B3) Trace one API endpoint route -> query/model -> response schema

Recommended endpoint: `GET /programs`.

Trace across: - Route/controller: `carms/api/routes/programs.py`. - Data access/session dependency: `carms/api/deps.py`, `carms/core/database.py`. - SQLModel models: `carms/models/gold.py`. - Response contracts: `carms/api/schemas.py`.

Then repeat quickly for: - `GET /programs/{program_stream_id}` - `GET /disciplines` - `POST /pipeline/run`

Step B4) Draw data flow on paper (or digital whiteboard)

Include exactly three lanes: 1. **Source lane:** files in `data/`. 2. **Transform lane:** Dagster assets/checks. 3. **Serve lane:** FastAPI routes + map/static artifacts.

Also annotate: - Where schema migrations apply. - Where indexes matter. - Where auth/rate limiting intercept requests.

Phase C — Runtime understanding (1–2 hours)

Step C1) Run tests and inspect failure behavior by breaking one transform locally

Workflow: 1. Run baseline tests. 2. Intentionally break one silver transform rule (e.g., province parsing or validity flag behavior). 3. Re-run tests. 4. Capture which tests fail and why. 5. Revert your break.

Purpose: develop confidence in what the tests protect vs what they do not.

Step C2) Start full stack with Docker and hit endpoints

Use full platform mode so Dagster + Postgres + API are all live.

Call at least: - `GET /health` - `GET /programs?province=ON&limit=5` - `GET /programs/{program_stream_id}` - `GET /disciplines` - `GET /map/data.json` - `POST /pipeline/run`

Capture examples of: - Normal 200 response. - At least one 4xx validation/auth/rate-limit response.

Step C3) Observe logs and connect pipeline steps to API outputs

In your notes, create a "cause/effect" map: - Which materialized table changed. - Which endpoint payload changed. - Which field in response is sourced from which table/column.

Phase D — Improvement readiness (1–2 hours)

Step D1) Identify one quality check to add

Examples: - Assert no duplicate `program_stream_id` in gold. - Assert `province` is in the allowed code set. - Assert description text is non-empty for active programs.

For interview: explain expected false positives/negatives.

Step D2) Identify one indexing/performance improvement

Options: - Add trigram index for `ILIKE` substring filters on discipline/school. - Add composite index for common combined filters. - Introduce materialized search view for high-frequency queries.

For interview: include expected query pattern and why b-tree alone may be insufficient for `%...%`.

Step D3) Identify one DS feature extension

Choose one and define MVP: - Program similarity search over descriptions (embedding + pgvector). - Preference-aware ranking (province + discipline + quota + text relevance). - Scenario simulation (e.g., discipline demand shifts by province).

For interview: include training/validation data assumptions and fairness caveats.

Step D4) Prepare your "first 30 days" plan

Week 1: Environment setup, lineage validation, endpoint smoke tests.

Week 2: Add one quality check + one observability improvement (logging/metrics).

Week 3: Ship one performance improvement with before/after benchmark.

Week 4: Prototype DS extension and present tradeoffs/next steps.

Deep-understanding checklist (self-evaluation)

You are interview-ready when you can answer "yes" to all:

- I can explain bronze/silver/gold using one real row example.
- I can describe exactly how `/programs` applies filters and pagination.
- I can name where auth and rate limiting are enforced.
- I can explain one migration and one index from the codebase.
- I can run the stack and troubleshoot a failed endpoint quickly.
- I can propose one quality and one performance enhancement with rationale.
- I can describe one DS extension with realistic implementation scope.

Suggested final deliverables for interview prep folder

Create a personal folder (outside git if preferred) containing: 1. `architecture-summary.md` (your 8–10 bullets). 2. `record-trace.md` (one row traced bronze->silver->gold). 3. `endpoint-trace.md` (route->query->schema walkthrough). 4. `runtime-findings.md` (tests, logs, failures, fixes). 5. `first-30-days.md` (execution plan + milestones).

If you can present these five artifacts clearly, you will demonstrate genuine end-to-end mastery rather than surface familiarity.