ETL Orchestrator Documentation

Rook IT ApS — v1.0.0

Rook IT ApS

Table of contents

1. ETL Orchestrator 6
1.1 What it does 6
1.2 Who it's for 6
1.3 Quick links 6
2. Getting Started 7
2.1 Getting Started 7
2.2 Installation 8
2.2.1 Requirements 8
2.2.2 Install 8
2.2.3 Final install step: upload the license 8
2.2.4 Customising port / data dir / container name 8
2.2.5 Upgrading 9
2.2.6 Lifecycle 9
2.2.7 What the installer does, in order 9
2.2.8 Uninstalling completely 9
2.3 Your First Pipeline 10
2.3.1 1. Create a pipeline 10
2.3.2 2. Add a step 10
2.3.3 3. Define an environment 10
2.3.4 4. Run 11
2.3.5 5. Make a change, re-run just the affected steps 11
2.3.6 6. Save a pipeline you're happy with 11
3. Demo (Synthmas) 12
3.1 Synthmas demo 12
3.1.1 What you'll build 12
3.1.2 Prerequisites 12
3.1.3 Steps 12
3.2 Setup 14
3.2.1 1. Request a trial license 14
3.2.2 2. Install the orchestrator 14
3.2.3 3. Upload the license 14
3.2.4 4. Import the Synthmas pipeline 15
3.2.5 5. (Optional) Switch on TLS 15
3.3 Vocabulary 16
3.3.1 What the bundle must contain 16

3.3.2 How to download 16
3.3.3 Upload 16
3.4 Source data 18
3.4.1 Option A — use the pre-generated bundle from the engine demos 18
3.4.2 Option B — generate your own with Synthea 18
3.4.3 Upload to the orchestrator 18
3.5 Run on CSV 19
3.5.1 Create the environment 19
3.5.2 Run it 19
3.5.3 Limitations of CSV 19
3.5.4 Where the data lands 20
3.6 Run on PostgreSQL 21
3.6.1 Prepare PostgreSQL 21
3.6.2 Create the environment 21
3.6.3 Run it — first time 22
3.6.4 "Reset" — wiping postgres externally 22
3.6.5 Confirm CSV / PostgreSQL parity 22
3.7 Reading the run-mode UI 23
3.7.1 The DAG 23
3.7.2 The ribbon bar 23
3.7.3 The per-step modal 23
3.7.4 Run history 24
4. Guide 25
4.1 Guide 25
4.2 Pipelines 26
4.2.1 Pipeline-level settings 26
4.2.2 Steps 27
4.2.3 Configure mode 27
4.2.4 Layout = topology 28
4.3 Environments 29
4.3.1 Source database 29
4.3.2 CDM database 29
4.3.3 Runtime settings 30
4.3.4 Secrets 30
4.3.5 Re-using config across envs 31
4.4 Source sets 32
4.4.1 Uploading 32
4.4.2 Filename conventions 32

4.4.3 Re-runs 32
4.5 Steps & RiaHs 33
4.5.1 Step kinds 33
4.5.2 Configuring an etl_map step 33
4.5.3 What "expected tables" buys you 33
4.5.4 Multi-RiaH wiring 33
4.6 Running a pipeline 35
4.6.1 Modes 35
4.6.2 Triggering a run 36
4.6.3 Live progress 36
4.6.4 Aborting 36
4.6.5 When a step fails 36
4.6.6 After the run 36
4.7 Re-running (cascade) 38
4.7.1 Two ways to trigger 38
4.7.2 When the cascade is just one step 38
4.7.3 When the cascade includes others 38
4.7.4 DAG colours during + after a cascade 39
4.7.5 Re-running after an orchestrator restart 39
4.7.6 When to use the full re-run instead 39
4.8 Save / Load 40
4.8.1 Pipeline `.etlopipeline` 40
4.8.2 Environment `.etloenv` 41
5. Reference 42
5.1 Reference 42
5.2 Bundle formats 43
5.2.1 `.etlopipeline` 43
5.2.2 `.etloenv` 43
5.3 Environment variables 45
5.3.1 Boot-time 45
5.3.2 Installer-only (host-side) 45
5.4 Endpoints 46
6. Concepts 47
6.1 Concepts 47
6.2 Re-run cascade — shared-table semantics 48
6.2.1 The problem 48
6.2.2 The right answer 48
6.2.3 CSV CDM cleanup, concretely 48

6.2.4 What survives across runs 49
6.2.5 Resume semantics 49
6.2.6 Single-flight, abort, and the cascade 49
6.3 DAG wiring 50
6.3.1 What the engine actually needs 50
6.3.2 How the orchestrator parses this 50
6.3.3 How the edges get drawn 51
6.3.4 What you don't need to specify 51
6.3.5 What the orchestrator does NOT auto-infer 52
6.4 CSV / PostgreSQL parity 53
6.4.1 Why it works now 53
6.4.2 How to verify 53
6.4.3 Pin engine 1.4.1+ 53
6.4.4 What stays asymmetric 54

1. ETL Orchestrator

A web app that orchestrates ETL Engine pipelines: upload Rabbit-in-a-Hat mapping files (RiaHs), wire them into a DAG by what they read and write, run them against an OMOP CDM, and watch every step unfold live.

The orchestrator replaces the hand-written shell scripts and Excel sheets that glue multi-RiaH ETL loads together today. One web UI for configure + run + history.

Get started → **Build your first pipeline →**

Download docs (PDF)

1.1 What it does

- **Pipeline configuration.** Upload one RiaH per step, attach any Usagi / CSVMap lookup files, set step ordering. The orchestrator auto-wires the DAG by reading each RiaH — `GetCDMTableId(@table)` becomes a consume edge to whichever sibling step stores `index/<table>`; multiple writers of the same `globalSeq/<seq>` get serialized so they don't race.
- **Environments.** Each `(pipeline, env)` pair is a separate target: source DB, CDM DB, license, log level, per-env `etl.conf` overrides, engine env vars + image overrides. Run the same pipeline against `dev` and `prod` with different connections — no copy-paste.
- **Running.** Live DAG view with per-step status colours, per-table row counts streaming as the engine emits them, full engine log capture, abort signal that crosses worker boundaries.
- **Selective re-run.** Re-run only the steps that share a CDM table with your target — the design-correct **re-run cascade**. Pre/post automation stay out of scope; unrelated dimension data is untouched.
- **Save / Load.** Export a pipeline (or an env) as a single bundle file (`.etlopipeline` / `.etloenv`) and load it on another instance — perfect for backup, sharing, or moving an env between machines.

1.2 Who it's for

People running OHDSI OMOP CDM loads from one or more source-data extracts, using the Rook IT ETL Engine to do the per-step mapping work. If today you hand-edit `run-N.sh` scripts and `etl.conf` files, this gives you a configure-once UI on top.

1.3 Quick links

- Install in 60 seconds
- Build your first pipeline
- Re-run cascade — what it is and why
- Bundle formats (`.etlopipeline`, `.etloenv`)

2. Getting Started

2.1 Getting Started

Two pages here:

1. **Installation** — get the orchestrator running on a host (docker or podman).
2. **First Pipeline** — create a pipeline, upload a RiaH, define an environment, run.

If you just want to play with the demo: install first, then in the UI hit **Pipelines → ⤒ Import** and upload the synthmas bundle that ships in the release.

2.2 Installation

The release ships as a single `.tar.gz` bundle containing the docker image, install + lifecycle scripts, and a README. Works with **docker** or **podman** out of the box.

2.2.1 Requirements

OS	Linux x86_64
Runtime	docker ≥ 20.10 **or** podman ≥ 4.0
Disk	~1 GB free for the image; data dir grows with usage (the synthmas demo data is ~ 500 MB)
Port	`8000` free (override via `ORCHESTRATOR_LISTEN_PORT`)

2.2.2 Install

```
tar xzf etl-orchestrator-X.Y.Z.tar.gz
cd etl-orchestrator-X.Y.Z
./bin/install-etl-orchestrator.sh
```

When it finishes you'll see a block with the URL and an **initial admin password** (random, generated once on first boot — copy it before clearing your terminal):

```
═══════════════════════════════════════════════════════
  ETL-Orchestrator 0.7.0 is running.
═══════════════════════════════════════════════════════
  URL:        http://your-host:8000/
  Local URL:  http://localhost:8000/

  Initial admin login:
    username: admin
    password: <random>
  (You'll be asked to change it on first sign-in.)
═══════════════════════════════════════════════════════
```

Open the URL, sign in as `admin` with the printed password, and you'll be forced through a password change before reaching the app.

2.2.3 Final install step: upload the license

The orchestrator boots and lets you sign in without a license, but it **can't actually run a pipeline** without one — every engine spawn needs a valid license file. So before you do anything else:

1. Make sure you have a `license.lic` file (request a trial at https://rook-it.com/licenses).
2. **Settings → License → Upload** → pick your `.lic` file.
3. The page now shows the licensed-to name + expiry.

That's the install done. You're now free to import / build pipelines, create environments, and run.

2.2.4 Customising port / data dir / container name

All three are env-var overrides on the installer:

```
ORCHESTRATOR_LISTEN_PORT=9000 \
ORCHESTRATOR_DATA_DIR=/srv/etlo \
```

```
ORCHESTRATOR_CONTAINER_NAME=etlo-staging \
    ./bin/install-etl-orchestrator.sh
```

The installer persists your chosen values to `bin/etlo.config` so the lifecycle script (`run-etl-orchestrator.sh`) finds the right container without you re-passing env vars.

2.2.5 Upgrading

Just re-run `./bin/install-etl-orchestrator.sh` with the newer tarball. The script removes the old container, loads the new image, restarts. Your data dir is untouched, so pipelines / envs / run history all carry over.

If the new release includes a schema change, the entrypoint's bootstrap step applies the alembic migration on first start automatically.

2.2.6 Lifecycle

After install, manage the container with:

```
./bin/run-etl-orchestrator.sh status      # container + /healthz
./bin/run-etl-orchestrator.sh logs        # follow logs
./bin/run-etl-orchestrator.sh stop        # data preserved
./bin/run-etl-orchestrator.sh start
./bin/run-etl-orchestrator.sh restart
./bin/run-etl-orchestrator.sh uninstall   # remove container; data + image kept
```

2.2.7 What the installer does, in order

1. **Detects docker or podman.** Hard error if neither is installed; clear error if the daemon isn't reachable from the current user (you need to be in the docker/podman group, or root).
2. `$tool load -i data/etl-orchestrator.docker` then re-tags as `:latest` in the same repo namespace the image landed in.
3. **Creates the data directory** (defaults to `/var/lib/orchestrator`).
4. **Probes the host for a container socket** (`/var/run/docker.sock`, `/run/podman/podman.sock`, `$XDG_RUNTIME_DIR/podman/podman.sock`) and bind-mounts the first hit to `/var/run/docker.sock` inside the container. The orchestrator uses it to spawn engine containers for each pipeline step.
5. **Removes any existing container of the same name** (the upgrade path).
6. `$tool run -d --restart unless-stopped` with port + volume + socket mounts; environment vars stripped to the essentials.
7. **Waits for** `/healthz` (up to 60 s) so failures surface immediately.
8. **Greps the bootstrap log for the initial admin password** and prints URL + password in the block above.
9. **Writes** `bin/etlo.config` with the chosen settings so the lifecycle script auto-loads them.

2.2.8 Uninstalling completely

```
./bin/run-etl-orchestrator.sh uninstall              # removes container
docker rmi rook-it.com/etl-orchestrator:latest       # frees ~ 300 MB
rm -rf /var/lib/orchestrator                          # DESTROYS ALL STATE
```

2.3 Your First Pipeline

End-to-end walkthrough — create a pipeline, add a step with a RiaH, define an environment, run.

2.3.1 1. Create a pipeline

Pipelines → + New.

Fields:

Field	Purpose
Name	Globally unique; appears in the menubar picker.
OMOP version	The CDM version every RiaH in this pipeline must target. Uploaded RiaHs are rejected if their `cdmDb.dbName` (`CDMV5.4` etc.) doesn't match.
Engine image	Default container image to spawn for steps (e.g. `etl-engine:latest`). Per-env override available later.

2.3.2 2. Add a step

Open the new pipeline (Configure mode). The DAG canvas shows the fixed **Pre-automation** and **Post-automation** nodes. Click **Add step**, name the step (e.g. `location`), and you land on its Configure page.

Upload three things (only the RiaH is mandatory):

- **RiaH** (`.json.gz`, exported from Rabbit-in-a-Hat). The orchestrator parses it to auto-populate:
- **expected tables** (every CDM table the RiaH writes via `tableToTableMaps`)
- **produces** (named `StoreSequence` / `StoreIndexMap` artifacts)
- **consumes** (cross-step `GetCDMTableId(@<table>)` lookups, resolved against sibling steps' `produces`)
- **Usagi** CSV(s) — vocab mapping lookups the RiaH's `UsagiMap` rules reference. Usagi files are plain CSV in a specific column layout (`sourceCode`, `sourceName`, `sourceFrequency`, `targetConceptId`, `mappingStatus`, …) — the same format the OHDSI Usagi tool exports.
- **CSVMap** csv(s) — plain CSV lookups the RiaH's `CSVMap` rules reference.

The DAG view updates the moment the upload succeeds: the new node appears, and any cross-step `consumes` show up as arrows from the producer to the new node.

2.3.3 3. Define an environment

Each pipeline targets one or more **environments** (= "where the source data lives + where the CDM lives + what license to use"). Add one:

Pipeline → Environments → + New.

Tabs:

- **Source database** — `csv` (a per-pipeline source set) or one of the RDBMS backends (postgres, mysql/mariadb, sqlserver). For RDBMS, fill in host/port/db/schema/user/password.
- **CDM database** — same shape; `csv` writes per-table CSV files to a per-env directory.
- **Settings** — log level, vocab cache size, worker threads, optional engine image override.

When source is `csv`, you'll need a **source set** — a named collection of CSV files at the pipeline scope. Upload them via **Pipeline → Source sets**.

2.3.4 4. Run

Switch the env's URL to Run mode (`/run`). Press ▶ **Run pipeline**. The DAG starts colouring per step:

- **white border** = not yet run.
- **blue** = currently running.
- **green** = success (every expected CDM table loaded).
- **amber** = partial (engine exited 0 but some expected tables didn't load).
- **red** = failed.
- **grey** = skipped (intentional — e.g. steps outside a re-run cascade, or `post_automation` when nothing it would produce is enabled).

The right-side strip lists currently-running steps with per-table progress bars. Each step's full state is one click away — tap the node, the modal opens with status, per-table row counts, the engine log (⬇ Log to download), and the **Rerun (cascade)** button.

2.3.5 5. Make a change, re-run just the affected steps

This is where the orchestrator earns its keep. Say your `conditions` RiaH had a bug and you've fixed + re-uploaded it. Right-click `conditions` in the DAG → **Rerun (cascade)**. A confirm dialog tells you exactly which steps will be re-run (the cascade — every step that shares a CDM table with `conditions`, transitively) and which CDM tables will be wiped before the re-run. Click through and only those steps run; everything else stays as it was.

→ How the cascade is computed.

2.3.6 6. Save a pipeline you're happy with

Configure → ⇩ Save. Downloads a `.etlopipeline` bundle (zip) containing the pipeline config, every step file (RiaH/Usagi/CSVMap, sha256-deduped), all wiring (resolved by step name so it re-links cleanly), and optionally the vocabulary zip. Hit **Pipelines → ⇧ Import** on another instance (or on the same one as a duplicate) and the bundle recreates the whole pipeline in one click.

→ Bundle formats.

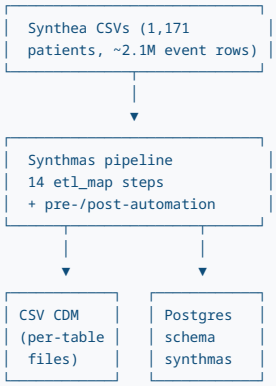
3. Demo (Synthmas)

3.1 Synthmas demo

End-to-end walk-through: get a running ETL Orchestrator with the Synthmas pipeline loading the Synthea synthetic-patient dataset into an OMOP CDM.

You'll do this **twice** — once with a CSV CDM target (self-contained, no database required) and once with a PostgreSQL CDM (closer to production: FK enforcement, downstream OHDSI tool compatibility). Both runs produce **row-for-row identical CDM tables** on engine 1.4.1+ — the parity isn't approximate, it's exact.

3.1.1 What you'll build



The demo covers all of:

- License upload + Settings tour.
- Athena vocab upload + what tables it must contain.
- Source CSVs (Synthea generation or download).
- CSV environment + run.
- PostgreSQL environment + run (same output, RDBMS-shaped for OHDSI tools + FK enforcement).
- Reading the DAG, the ribbon bar, the per-step modal.

3.1.2 Prerequisites

- A Linux host with **docker** or **podman** installed.
- Internet egress (initial image pull, Athena download, optional Synthea generation).
- For the PostgreSQL run: a reachable PostgreSQL ≥ 14 instance you can create a schema in (the demo creates `synthmas`).

3.1.3 Steps

1. **Setup** — request a temp license, install the orchestrator, log in, import the Synthmas pipeline.
2. **Vocabulary** — what Athena bundle you need + how to upload it.

3. **Source data** — get the Synthea CSVs.
4. **Run on CSV** — create the CSV env, kick off the pipeline, watch it complete.
5. **Run on PostgreSQL** — create the schema + the RDBMS env, run again.
6. **Reading the run-mode UI** — DAG colors, ribbon bar, per-step modal, what the numbers mean.

3.2 Setup

3.2.1 1. Request a trial license

The orchestrator drives the **ETL Engine**, which is licensed software. You need a license with the **ETL** feature to run the Synthmas demo (it uses `StoreIndexMap` + `StoreSequence` rules that aren't available in the free tier).

To get a **5-day free trial license**:

1. Go to **https://rook-it.com/licenses** and create an account.
2. Request your one-time 5-day trial. You'll receive a `license.lic` file by email.

If 5 days isn't enough — for example, you're evaluating a longer production pilot — write to **licenses@rook-it.com**. We'll arrange a short conversation about your use case first, then issue a longer license tailored to it. We don't issue extended licenses unattended, just so we can match the license terms to the work you're actually doing.

3.2.2 2. Install the orchestrator

Download the latest release tarball from your Rook IT release page (installer comes inside):

```
tar xzf etl-orchestrator-<version>.tar.gz
cd etl-orchestrator-<version>
sudo bin/install-etl-orchestrator.sh
```

The installer:

- Probes your host for docker / podman + their sockets.
- Loads the image from `data/etl-orchestrator.docker`.
- Starts the container with `--restart unless-stopped`.
- Waits for `/healthz` to come up.
- Prints **a randomly-generated initial admin password to stdout**. This is shown **once** — copy it somewhere safe before closing the terminal.

Default URL: `http://localhost:8000/`. Log in as `admin` with the random password the installer printed.

> **Different host port**
>
> Set `ORCHESTRATOR_LISTEN_PORT=8765` (or similar) in the environment before running `install-etl-orchestrator.sh` — both the container and the host port mapping pick it up.

3.2.3 3. Upload the license

This is the last install step before you can do anything else — without a license, the orchestrator boots and lets you in, but every engine spawn will fail with a clear "no valid license" error.

In the UI: **Settings → License → Upload** `license.lic`.

A valid license with the ETL feature unlocks the StoreIndexMap / StoreSequence rules every Synthmas RiaH uses. The license page shows the licensed-to name + expiry once it's installed.

3.2.4 4. Import the Synthmas pipeline

Download the pipeline bundle:

⤓ synthmas-pipeline-1.0.3.etlopipeline

In the UI: **Pipelines → Import** → select the `.etlopipeline` file → **Import**.

You should now see a "Synthmas" pipeline on the **Pipelines** page with 14 etl_map steps wired into a DAG of cross-step `StoreIndexMap` / `StoreSequence` edges.

3.2.5 5. (Optional) Switch on TLS

If the orchestrator is reachable beyond `localhost`, **Settings → TLS** to pick a mode:

- **Self-signed** for a quick lab cert (Rook IT branded; valid 13 months).
- **Self-managed** to upload a real cert + key + chain.

Either way the change takes effect on the next container restart (`bin/run-etl-orchestrator.sh restart`); gunicorn doesn't hot-reload TLS.

→ Next: **Vocabulary**.

3.3 Vocabulary

OMOP mappings (`VocabMap("SNOMED")` etc.) resolve source codes against the OMOP **Athena** vocabulary tables. You need to download those once and upload the zip to the orchestrator.

3.3.1 What the bundle must contain

The Synthmas mappings exercise these vocabularies:

Vocabulary	Used by
SNOMED	conditions, procedures, allergies, devices
LOINC	observations (clinical step), measurements
RxNorm	drug_exposure
CVX	drug_exposure (immunizations)

You can include more (LOINC Component Hierarchy, ATC, etc.) without hurting — the engine indexes the whole vocab on load and unused entries just sit there.

Required tables inside the Athena zip:

- `CONCEPT.csv`
- `CONCEPT_RELATIONSHIP.csv`
- `CONCEPT_ANCESTOR.csv`
- `CONCEPT_SYNONYM.csv`
- `CONCEPT_CLASS.csv`
- `VOCABULARY.csv`
- `RELATIONSHIP.csv`
- `DOMAIN.csv`
- `DRUG_STRENGTH.csv`

These are exactly the file set the engine's `vocabulary_loader` expects (filename prefix `vocabulary_download_v5_*.zip`).

3.3.2 How to download

1. Create an account on **https://athena.ohdsi.org/**.
2. **Download** → tick the vocabularies above → start the download.
3. Athena emails you when the bundle is ready (usually within an hour).
4. Click the email link, save the resulting `.zip` (typically named `vocabulary_download_v5_<date>_<id>.zip`).

Total uncompressed size: ~3.3 GB; compressed ~600 MB. The orchestrator keeps it as a single blob and never re-extracts at runtime.

3.3.3 Upload

In the UI: **Pipelines → Synthmas → Configure → Vocabulary** → upload the zip. Give it a version string (the Athena release date works: `v5_2026-01-15`). The orchestrator computes a content hash and won't re-extract if the same content is uploaded again.

Notes:

- The vocab is **pipeline-scoped** — once uploaded, every env using this pipeline shares it. Switch vocab by uploading a new version; envs see the change on their next pre-automation run.
- Loading time into the postgres CDM is **~10 minutes for the full Athena bundle** on a modest box; the CSV CDM path doesn't load into a database — it serves the vocab files directly from disk.

→ Next: **Source data**.

3.4 Source data

The Synthmas demo runs against **Synthea** synthetic-patient CSVs — open data with no PHI, perfect for demos. You need ~1,000 patients (the default Synthea run produces 1,171 with the recommended seed).

3.4.1 Option A — use the pre-generated bundle from the engine demos

Easiest: the etl-engine repo ships a small Synthea bundle under `demos/synthmas-single-RiaH/etc/etl-engine.d/db/`. Copy those CSVs into a local folder:

```
git clone https://github.com/rook-it/etl-engine
cp -r etl-engine/demos/synthmas-single-RiaH/etc/etl-engine.d/db ./synthea-source
ls synthea-source/
# allergies.csv  careplans.csv  claims.csv  claims_transactions.csv
# conditions.csv  devices.csv  encounters.csv  imaging_studies.csv
# immunizations.csv  medications.csv  observations.csv
# organizations.csv  patients.csv  payer_transitions.csv  payers.csv
# procedures.csv  providers.csv  supplies.csv
```

3.4.2 Option B — generate your own with Synthea

If you want a larger / different cohort:

```
# Synthea v3.x — uses Java 11+
git clone https://github.com/synthetichealth/synthea
cd synthea
./run_synthea -p 1000 -s 42 \
              --exporter.csv.export true \
              --exporter.csv.folder_per_run false \
              --exporter.fhir.export false
mv output/csv ../synthea-source
```

`-p 1000` = 1,000 living patients (plus deceased who lived during the sim window); `-s 42` = deterministic seed; the `--exporter.fhir.export false` flag skips the large FHIR JSON output we don't use.

3.4.3 Upload to the orchestrator

In the UI: **Pipelines → Synthmas → Configure → Source sets → New** → name it (`synthea-1k`), upload the CSVs (multi-select the whole folder).

You can keep multiple source sets per pipeline (different patient cohorts / different Synthea releases) and pick which one an env reads from on its environment page.

→ Next: **Run on CSV**.

3.5 Run on CSV

The CSV target is the fastest way to see the full pipeline run end to end. No database to set up; the engine writes per-table CSV files to disk and serves the vocab from the Athena bundle directly.

> **Pin engine 1.4.1 or newer**
>
> Engine 1.4.1 made the CSV target semantically identical to the RDBMS targets for CDM-lifecycle management (canonical OMOP headers via `create omop cdm: once`, truncate-to-canonical via `clean omop data: true`, virtual-field-aware header validation, append-not- truncate `loadWriteTable`). Earlier engines have known correctness gaps on multi-writer CDM tables (e.g. `cost` written by both `synthmas-claims` and `synthmas-cost-drugs`). The orchestrator's settings page lets you pin a per-pipeline engine image; CSV demo envs should pin `rook-it.com/etl-engine:1.4.1` or later.

3.5.1 Create the environment

Pipelines → Synthmas → Environments → New:

Field	Value
Name	`csv-demo`
Source DB backend	CSV
Source set	the `synthea-1k` source set you uploaded
CDM DB backend	CSV
Log level	`info`
Vocab cache size	`10000` (default)
Worker threads	`10` (default)

Save. The orchestrator stores the env config; nothing happens yet.

3.5.2 Run it

On the env page, click ▶ **Run pipeline**. You should see:

1. **Pre-automation** turns blue (running), then green. On CSV the engine writes each `<cdm-out>/<table>` with its canonical OMOP header (so multi-writer tables like `cost` see the full column set regardless of which step writes first), truncates any existing tables back to that canonical header (SQL TRUNCATE semantics, kept schema), and the orchestrator extracts the Athena vocab CSVs into the same dir. Fast — usually under a second once the vocab is extracted.
2. Each etl_map step turns blue in topo order — location → care_site → provider → person → visits → conditions → claims → clinical → allergies → procedures → drugs → devices → coverage → cdm-source.
3. **Post-automation** turns blue and then green: on CSV it generates `observation_period` (one row per person, via the engine's driver-agnostic builder). Eras + preceding-visit-IDs stay skipped on CSV — see Limitations of CSV.

Total wall-clock: ~10 minutes on a typical laptop (1,171 patients).

3.5.3 Limitations of CSV

CSV is great for the demo — no DB to set up, file-by-file inspectable output — but it has structural limits you'll hit on real loads.

observation_period: yes; eras + preceding-visit-IDs: still RDBMS-only

`observation_period` **is** generated on CSV — the engine ships a driver-agnostic post-automation builder that folds each person's clinical events into one period (MIN start … MAX end). The Synthmas demo relies on this (the old `careplans` / `coverage` `observation_period` writes were removed, and the `careplans` step with them), so post-automation runs and turns green on CSV.

Eras and preceding-visit-IDs are not produced on the CSV demo:

- `condition_era` , `drug_era` , `dose_era` are **not created** — the orchestrator forces the era toggles off for a CSV target (lifting that is a follow-up).
- `visit_occurrence.preceding_visit_occurrence_id` stays `NULL` for every row.

If your downstream analysis needs eras or visit-graph navigation, switch to an RDBMS target — the same pipeline, run against a postgres env, produces both.

No referential integrity, no downstream tooling

The engine can't enforce FK constraints across CSV files at write time — a `condition_occurrence.person_id` pointing at a non-existent `person.person_id` will land on disk without complaint. And tools that expect a real OMOP database (Atlas, HADES, the OHDSI Data Quality Dashboard) can't connect to a folder of CSVs. For both reasons CSV is a demo / inspection target, not a production destination.

3.5.4 Where the data lands

CSV CDM tables live under `/var/lib/orchestrator/csv_cdm/<env_id>/<table>` . One file per CDM table, comma-separated, with the CDM header line at the top.

```
docker exec etl-orchestrator-orchestrator-1 ls /var/lib/orchestrator/csv_cdm/2/
# care_site  condition_occurrence  cost  death  device_exposure
# drug_exposure  location  measurement  observation  observation_period
# payer_plan_period  person  procedure_occurrence  provider
# visit_detail  visit_occurrence  cdm_source  (plus Athena vocab files)
```

Spot-check a multi-writer table to confirm the canonical-header behavior is working — `cost` is written by both `synthmas-claims` and `synthmas-cost-drugs` , so its header has to carry every column either writer mentions:

```
docker exec etl-orchestrator-orchestrator-1 \
  head -1 /var/lib/orchestrator/csv_cdm/2/cost | tr ',' '\n' | nl
#    1 cost_id
#    2 cost_event_id
#    3 cost_domain_id
#    4 cost_type_concept_id
#  ...
#   21 drg_concept_id            ← from synthmas-cost-drugs
#   22 drg_source_value          ← from synthmas-cost-drugs
```

22 columns, including the cost-drugs-only ones — that's the canonical OMOP 5.4 `cost` schema, laid by the engine in `pre_automation` . On older engines (pre-1.4.0) the file's header had only the first writer's columns and `synthmas-cost-drugs` would abort with "field `drg_concept_id` is missing in table".

→ Next: **Run on PostgreSQL** — the same pipeline, RDBMS target, **row-for-row identical CDM output** on engine 1.4.1+.

3.6 Run on PostgreSQL

The RDBMS target is the production-shaped OMOP CDM: FK constraints enforced at write time, downstream OHDSI tools (Atlas, HADES, the Data Quality Dashboard) can connect to it as-is. Setup is a bit more work than CSV (you need a postgres instance + a schema), but the CDM-side output is **identical row-for-row** to the CSV run on engine 1.4.1+ — the difference is the destination, not the data.

3.6.1 Prepare PostgreSQL

You need PostgreSQL ≥ 14. On the postgres host (or any client that can reach it):

```
psql -h <pg_host> -U postgres -d postgres
```

```
-- A dedicated user is best practice; for the demo the postgres super
-- user works fine. Whichever you use, the role needs CREATE on the
-- target schema.
CREATE SCHEMA synthmas AUTHORIZATION postgres;

-- (Optional) verify
SELECT current_schema(), current_user;
```

The orchestrator doesn't create the schema — you do. It does create **every table inside the schema** + indexes + constraints during pre-automation.

3.6.2 Create the environment

Pipelines → Synthmas → Environments → New:

Field	Value
Name	`pg-demo`
Source DB backend	CSV (same as before — Synthea source)
Source set	`synthea-1k`
CDM DB backend	**PostgreSQL**
Host / Port / Database	your postgres connection
User / Password	the role above
Schema	`synthmas`
Vocab cache size	`10000`
Max DB connections	`50` (lower if your postgres is small)

Save. The orchestrator stores the connection (the password is encrypted at rest with your login password; you'll be prompted to re-enter it once per session before runs).

3.6.3 Run it — first time

▶ **Run pipeline**. Pre-automation now does real work:

1. **Build CDM** (creates every OMOP table in your `synthmas` schema). Fast — ~1 second.
2. **Populate vocabulary** (loads the Athena bundle into the `concept`, `concept_relationship`, etc. tables). **~10 minutes on a typical box** for the full Athena bundle, dropping indexes first for bulk-load speed.
3. **Add constraints + indexes back**. Another ~10 minutes. The pre-auto modal shows "Indexes: X of 17" as it goes.

Total pre-automation: **~20 minutes the first time**. Subsequent runs skip the load entirely — see **Settings → environment → CDM state**; once vocab is loaded for an env, the orchestrator records it and re-runs short-circuit pre_auto's vocab phase.

After pre-auto, etl_map steps + post-auto run the same as on CSV, just faster per step thanks to indexed vocab lookups. Total wall-clock for a **first** run: ~30–35 minutes; **subsequent** runs (after vocab is cached): ~5 minutes.

3.6.4 "Reset" — wiping postgres externally

If you `DROP SCHEMA synthmas CASCADE; CREATE SCHEMA synthmas;` (e.g. to re-test from scratch), the orchestrator's cache still thinks the schema + vocab are loaded. Click **Edit env → CDM state → Reset CDM state** to clear the cache; the next run will recreate the schema + reload vocab.

3.6.5 Confirm CSV / PostgreSQL parity

On engine 1.4.1+, the two runs you just did should produce row-for-row identical CDM tables. Spot-check it from the orchestrator container:

```
docker exec etl-orchestrator-orchestrator-1 python <<'PY'
from collections import defaultdict
from app import create_app
from app.models import PipelineRun, StepRun, StepRunTable
app = create_app()
with app.app_context():
    # Replace with your two PipelineRun ids — one CSV, one PG.
    rids = (193, 194)
    per_run = {}
    for rid in rids:
        sr_ids = [sr.id for sr in StepRun.query.filter_by(pipeline_run_id=rid).all()]
        per = defaultdict(int)
        for srt in StepRunTable.query.filter(StepRunTable.step_run_id.in_(sr_ids)).all():
            per[srt.table_name] += srt.rows_added or 0
        per_run[rid] = per
    tables = sorted({t for d in per_run.values() for t in d})
    print(f'{"table":<24}', '  '.join(f'{r:>10}' for r in rids), '  delta')
    for t in tables:
        a, b = per_run[rids[0]][t], per_run[rids[1]][t]
        print(f'{t:<24}  {a:>10}  {b:>10}  {b-a:>+6}')
PY
```

Every `delta` column should be `+0`. A typical synthmas demo lands at 2,015,051 total CDM rows across 16 tables, identical CSV vs PG. If deltas show up, check that both runs used engine 1.4.1+ — older engines have known CSV-side correctness gaps (see CSV / PostgreSQL parity).

→ Next: **Reading the run-mode UI**.

3.7 Reading the run-mode UI

Three surfaces tell you what's going on during a run.

3.7.1 The DAG

The pipeline view in run mode renders every step as a node, colored by state. During an in-flight run:

Color	Meaning
Green	Step completed successfully in this run, OR has valid prior-run data that's not part of this re-run's scope.
Blue	Step currently executing.
White	Step is queued/stale: the dispatcher hasn't reached it yet (full re-run), OR the cascade includes it but it hasn't started, OR its prior data is potentially stale because something it depends on (FK / shared writer / artifact wiring) is being re-run.
Grey dotted	Step skipped — either because it's not in the cascade scope, or it's intentionally inert (e.g. a step with no expected tables).

Hover any node to see its expected tables. Right-click for a context menu:

- **etl_map** steps → ↻ **Rerun (cascade)** with a preview showing which other steps will also re-run (shared-table writers, transitively).
- **post-automation** → ↻ **Re-run this step** (single-step pipeline_run).
- **pre-automation** has no right-click rerun — equivalent to ▶ Run pipeline.

3.7.2 The ribbon bar

The big bar next to the timer shows overall progress as a percentage of "the work this run intends to do" — i.e. everything in `will_run_set`. For a full re-run that's every step; for a cascade re-run that's just the cascade members; for a single-step re-run that's just one step.

The number is honest in the sense that pre-auto and post-auto have a synthetic rows-equivalent budget (so the bar moves during their silent vocab-load / index-restore / era phases); they contribute their share of the percentage but the strip cards beneath show just `(82%)` rather than a misleading "822k / 1M rows".

3.7.3 The per-step modal

Click any step node to open its modal. Three branches by status:

Running

- For **etl_map** steps: a per-expected-table progress list with rows loaded vs scan-based / calibrated estimate, plus a "current table" pointer showing what the engine is mapping right now. Click a table row to open its engine log (live-tailing while the step runs — auto-refreshes every 2 s, preserves your scroll position so you can read older lines mid-stream).
- For **pre-automation** with vocab loading: "Loaded records" table fills in as each vocab table completes (`concept_ancestor` : 38,832,744 etc.), then an "Indexes: N of 17 added" progress bar for the index-restore phase.
- For **post-automation**: `observation_period` (one row per person — ~1,162 on the synthea-1k set) on both CSV and RDBMS, plus, on an RDBMS target only, era counts as each task completes (`condition_era` : 46,187 records, `drug_era` : 9,500, `dose_era` : 26,661) and preceding-visit-IDs counts.

Finished (success / partial / failed / skipped)

- Per-table outcomes table: rows_loaded, rejected (with a sample message hinting at why), per-table error kind.

- A "↻ Rerun cascade" button on etl_map steps that re-opens the preview + confirm flow.
- A download link for the per-step engine log (gzipped, named after the step + run id).

Queued / not in scope

- Static configuration (current RiaH filename, expected tables) and a message explaining why the run hasn't touched this step.

3.7.4 Run history

The "Recent runs" table beneath the DAG auto-updates as the in-flight run progresses — no page reload needed. Each row links to a per-run detail page with the full step-by-step outcome history and downloadable engine logs.

That's the whole tour. From here, look at the Concepts section if you want to understand WHY the cascade walks shared-table writers transitively, or Reference for the bundle formats and HTTP endpoints.

4. Guide

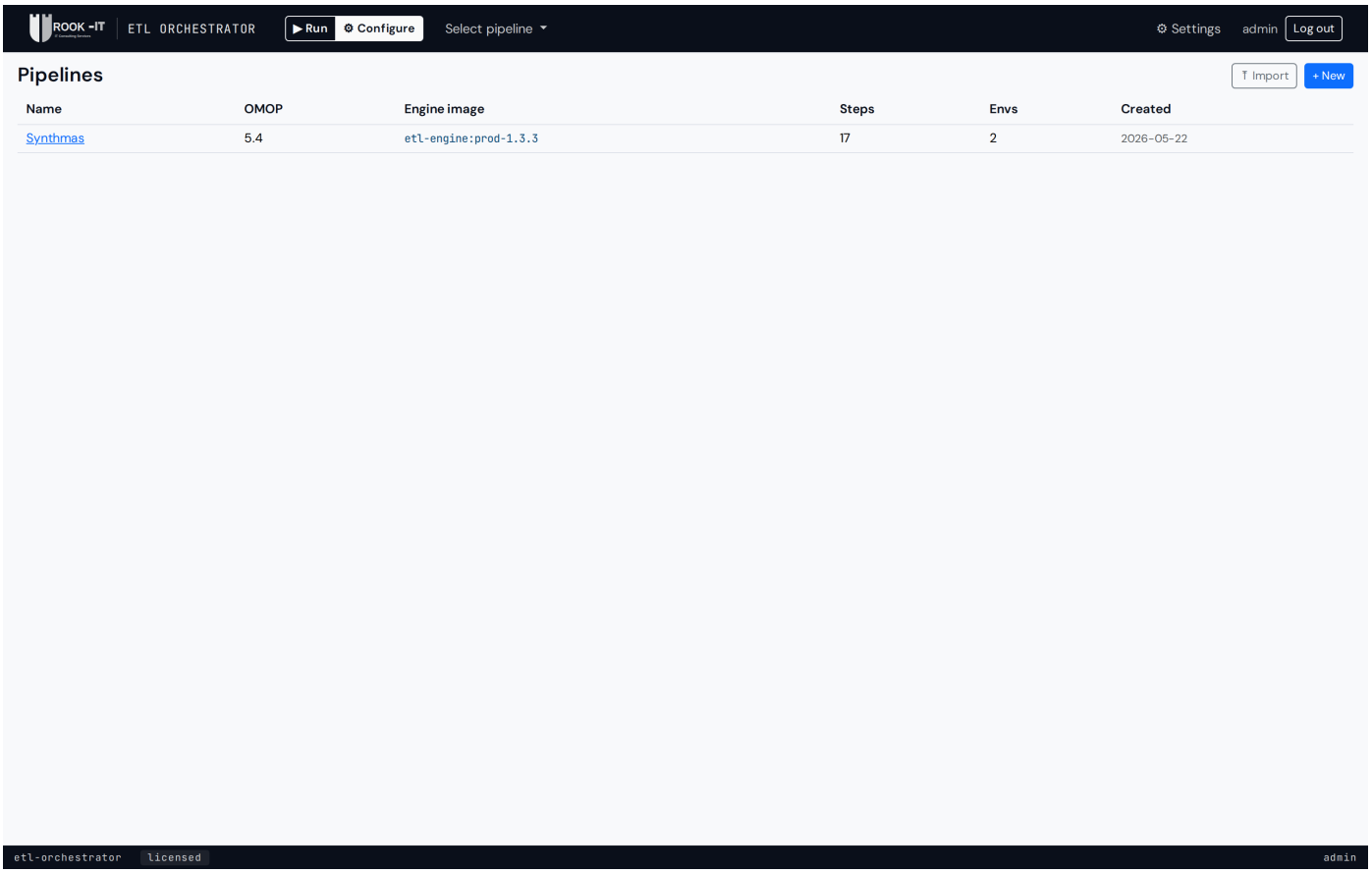
4.1 Guide

Topic-by-topic reference for the day-to-day work.

- **Pipelines** — what a pipeline is, the configure-mode DAG, step ordering, fixed automation steps.
- **Environments** — per-env source / CDM connections, settings, runtime knobs.
- **Source sets** — uploading and managing source CSV files when the source backend is `csv`.
- **Steps & RiaHs** — step kinds, uploading RiaH / Usagi / CSVMap versions, what the orchestrator parses out of a RiaH.
- **Running a pipeline** — kicking off a run, live progress, abort.
- **Re-running (cascade)** — the cascade model and how to trigger from the DAG or modal.
- **Save / Load** — `.etlopipeline` and `.etloenv` bundles.

4.2 Pipelines

A **pipeline** is the recipe: which CDM version to load into, which mapping files (RiaHs) describe the source → CDM transformation, and how the steps wire together. It owns no data of its own — environments attach to it and run it.



Pipelines index — OMOP version, pinned engine image, step + env counts at a glance.

4.2.1 Pipeline-level settings

Setting	What it does
Name	Just for you — pick anything unique within the orchestrator.
OMOP version	`5.3` , `5.4` , or `6.0` . The engine creates the matching schema during pre-automation. All RiaHs uploaded later must match (the configure page rejects a mismatch).
Engine image	Defaults to whatever the global default is in Settings. Override per-pipeline to pin a specific engine release for reproducibility.
Vocabulary	Upload an Athena bundle here; every env using this pipeline shares it.
Post-automation toggles	`create observation periods` , `add preceding visit IDs` , `create condition / drug / dose eras` — engine features that run after the etl_map steps. Toggle off the ones you don't need. (`observation_period` is generated on both CSV and RDBMS; the others are RDBMS-only.)
Description	Free-text. Shown in the pipelines list.

4.2.2 Steps

A pipeline has three kinds of step:

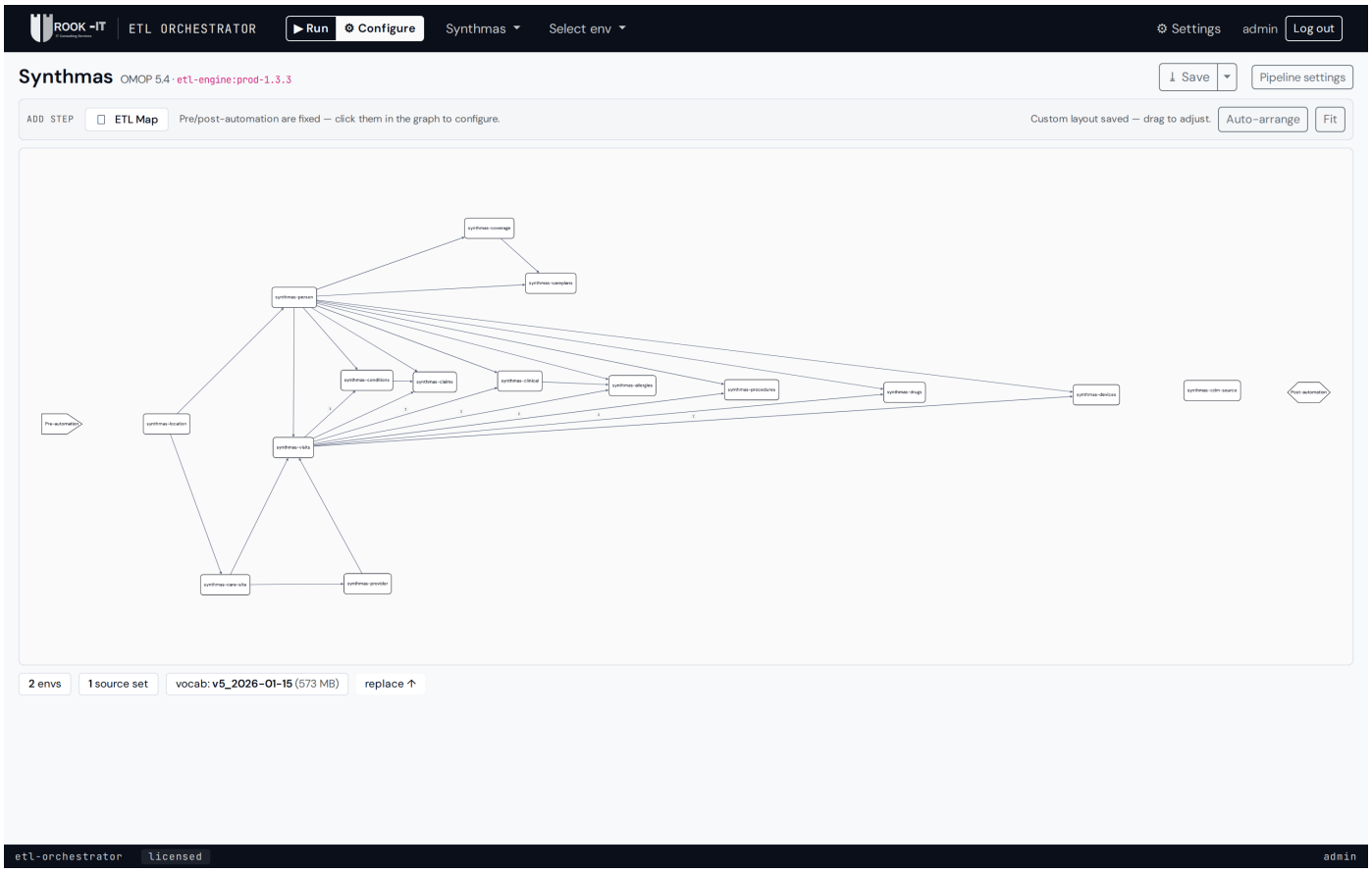
- **Pre-automation** (fixed, one per pipeline) — builds the CDM schema
- loads vocab. The orchestrator seeds this automatically; you don't delete or reorder it.
- **etl_map** (you create these) — the actual mapping work. One RiaH per step, plus optional Usagi / CSVMap lookup files. You can have as few or as many as you want.
- **Post-automation** (fixed, one per pipeline) — observation_period, eras + preceding-visit-IDs. Toggled by the pipeline-level flags.

Pre-automation always runs first, post-automation always runs last. Between them, etl_map steps run in **topological order** derived from the wiring (next section).

4.2.3 Configure mode

The configure-mode view of a pipeline is a Cytoscape DAG:

- Nodes are steps. Shape encodes kind (rectangle = etl_map, tag-shape = pre-automation, hexagon = post-automation).
- Solid edges are **artifact wiring** — one step's `StoreIndexMap` / `StoreSequence` output feeds another step's `GetCDMTableId` / `AutoGenId` consumer. The orchestrator detects these automatically by parsing the RiaHs.
- Dashed edges are **FK awareness** — derived from the OMOP CDM itself, shown as supplemental context. Not gated on; just a "you should know about this" hint.



Configure mode for the Synthmas pipeline: pre-automation pinned on the left, post-automation on the right, etl_map steps arranged by the topology the orchestrator inferred from RiaH artifact wiring. Drag nodes to reshape; the layout persists.

Drag nodes to lay out the DAG manually — the orchestrator persists the layout per pipeline.

4.2.4 Layout = topology

Step ordering on screen is the dispatcher's actual execution order. Drag a step to a different position and the next run honours it; the orchestrator only refuses an arrangement that violates a hard topo constraint (a consumer can't be moved before its producer).

4.3 Environments

An **environment** binds a pipeline to a concrete source database, a concrete CDM target database, and a set of runtime knobs. A pipeline can have many environments — typically at least one for development (CSV target, fast feedback) and one for production (postgres or similar).

4.3.1 Source database

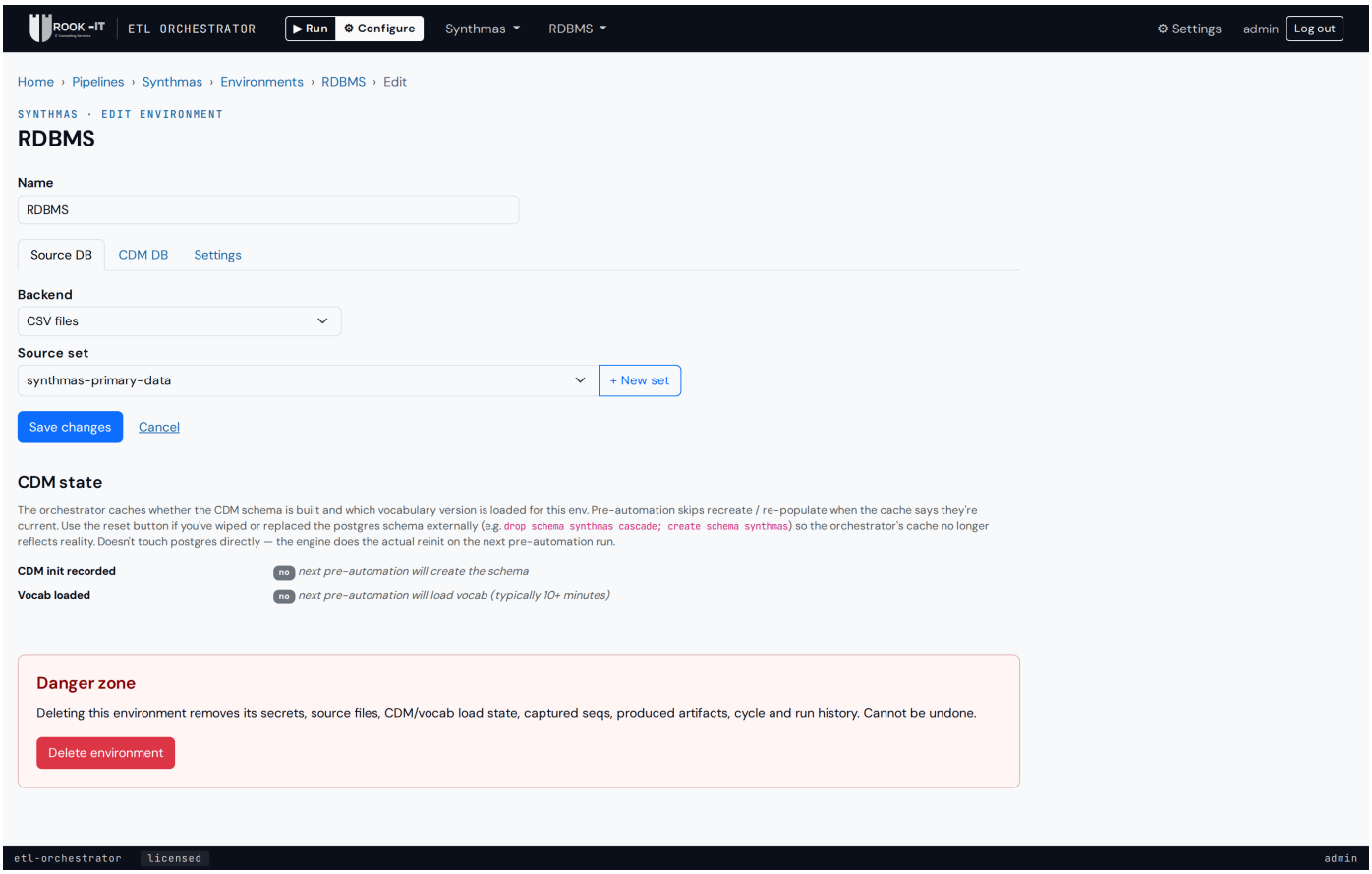
Backend	Use when
CSV	Your source is a folder of CSV files (Synthea, an upstream extract). You upload them as a "source set" on the pipeline; the env picks one to read from.
PostgreSQL / MariaDB / SQL Server	Your source lives in a real database. Provide host / port / db / schema / user / password.

The engine adapts its read path to the backend; mappings (RiaHs) don't care.

4.3.2 CDM database

Same backends. Choose by audience:

- **CSV** when you want to inspect output as files (per-table CSVs land under `/var/lib/orchestrator/csv_cdm/<env_id>/`). Good for demos, quick correctness checks, and cross-checking an RDBMS env's output for free (the CDM tables are row-for-row identical on engine 1.4.1+).
- **PostgreSQL** (and friends) when you want referential-integrity enforcement at write time, and a target that downstream OHDSI tools (Atlas, HADES, the Data Quality Dashboard) can connect to as-is.



Environment edit page — source + CDM backends are picked at the top; runtime knobs follow. Secrets are entered separately and encrypted at rest with your login password.

4.3.3 Runtime settings

Setting	Default	Note
Log level	`info`	Engine spdlog level. `debug` is verbose but useful when investigating mapping issues.
Vocab cache size	`10000`	In-memory cache of resolved concept IDs. Bump for large vocabularies / wider data.
Max CDM connections	`50`	Size of the engine's postgres connection pool. Match to what your postgres allows.
Worker threads	`10`	Number of engine worker threads per step. Rule of thumb: `max CDM connections + CPU cores` — threads block on DB writes, so over-subscribing CPU is fine as long as connections are available.
Engine image override	empty	Pin a specific engine release for this env (otherwise the pipeline default is used).

4.3.4 Secrets

Database passwords are encrypted at rest with your **login password** (field-level encryption). The orchestrator never persists your login password — instead the first action that needs a secret in a session prompts you to re-enter it, then caches it in memory for the rest of the session.

If you forget your login password, the encrypted secrets are unrecoverable — you'll re-enter the connection passwords once and move on. Everything else (pipeline config, env settings, run history) is unaffected.

4.3.5 Re-using config across envs

When you've built one env, you can **Save** it (`.etloenv` file) and **Load** elsewhere. See Save / Load for the details + what's included/ excluded.

4.4 Source sets

When an environment's **source database backend is CSV**, you don't configure a database — you point at a **source set**, which is a named collection of source CSV files attached to the pipeline.

You can have multiple source sets per pipeline (e.g. `synthea-1k-2024-01`, `synthea-10k-2024-06`), and pick which one an env reads from on its environment page. Switching cohorts is a one-click operation.

4.4.1 Uploading

Pipelines → \<your pipeline> → Source sets → New → name the set → upload the CSV files. The orchestrator content-addresses each file by sha256 — re-uploading identical content costs no extra disk.

Each file shows up with size, upload time, and a delete button. You can update individual files later without re-uploading the whole set.

4.4.2 Filename conventions

The engine reads files by name. For Synthea, the orchestrator expects the standard names (`patients.csv`, `encounters.csv`, `organizations.csv`, etc.). If your upstream uses different names, either rename before upload or adjust the RiaH's source table names to match what you've got.

4.4.3 Re-runs

A source set is a per-pipeline reusable asset. Switching the env's source set between runs lets you re-run the pipeline on a different cohort without rebuilding any of the mapping config — useful for "does it still load cleanly when I throw 10× the patients at it" tests.

4.5 Steps & RiaHs

A **step** is one unit of mapping work inside a pipeline. The actual mapping rules live in a **RiaH** — a gzipped JSON file produced by Rabbit-in-a-Hat (the OHDSI tool that lets you author CDM mappings visually).

4.5.1 Step kinds

Kind	Owner	Editable
etl_map	You	Yes — create, rename, reorder, upload RiaH versions.
pre_automation	Orchestrator	Toggles only (pipeline-level vocab + create-CDM are fixed on; logging knobs etc. are env-level).
post_automation	Orchestrator	Toggle observation_period + individual eras + preceding-visit-IDs at the pipeline level.

Pre- and post-automation are created automatically when you make a pipeline; you can't delete or reorder them.

4.5.2 Configuring an etl_map step

Pipelines → \<your pipeline> → \<step> → Configure:

- **RiaH** — the mapping file. Upload a `.json.gz` from Rabbit-in-a-Hat (or RiaH-style `.json.gz` produced by any compatible tool). On upload the orchestrator parses it and surfaces:
- **CDM version** — must match the pipeline's `omop_version`, otherwise upload is rejected.
- **Expected tables** — every CDM table the RiaH writes (derived from `tableToTableMaps`).
- **Produces** — named `StoreSequence` + `StoreIndexMap` artifacts.
- **Consumes** — cross-step `GetCDMTableId(@<table>)` lookups resolved against sibling steps' produces.
- **Usagi files** — CSV lookup tables in OHDSI Usagi format (`sourceCode`, `sourceName`, `sourceFrequency`, `targetConceptId`, `mappingStatus`, `equivalence`, `statusSetBy`, `statusSetOn`, …). Referenced by `UsagiMap` rules inside the RiaH.
- **CSVMap files** — plain CSV lookups. Referenced by `CSVMap` rules.

Every file is **versioned** — upload a new one and the prior version moves to "archived" status. Each version has Restore + Delete buttons, plus a download button so you can pull the file back out of the orchestrator to edit locally.

4.5.3 What "expected tables" buys you

The expected-tables list is the authoritative answer to "did this step load everything it was supposed to" — the run-mode modal shows per-table outcomes (loaded / error / missing) against it, and the orchestrator downgrades the step verdict from `success` to `partial` if any expected table came back with zero rows.

If the auto-derived list is wrong (e.g. you have a RiaH that uses `InsertRecord` to seed a constant-row table without `tableToTableMaps`), edit the list manually on the step configure page.

4.5.4 Multi-RiaH wiring

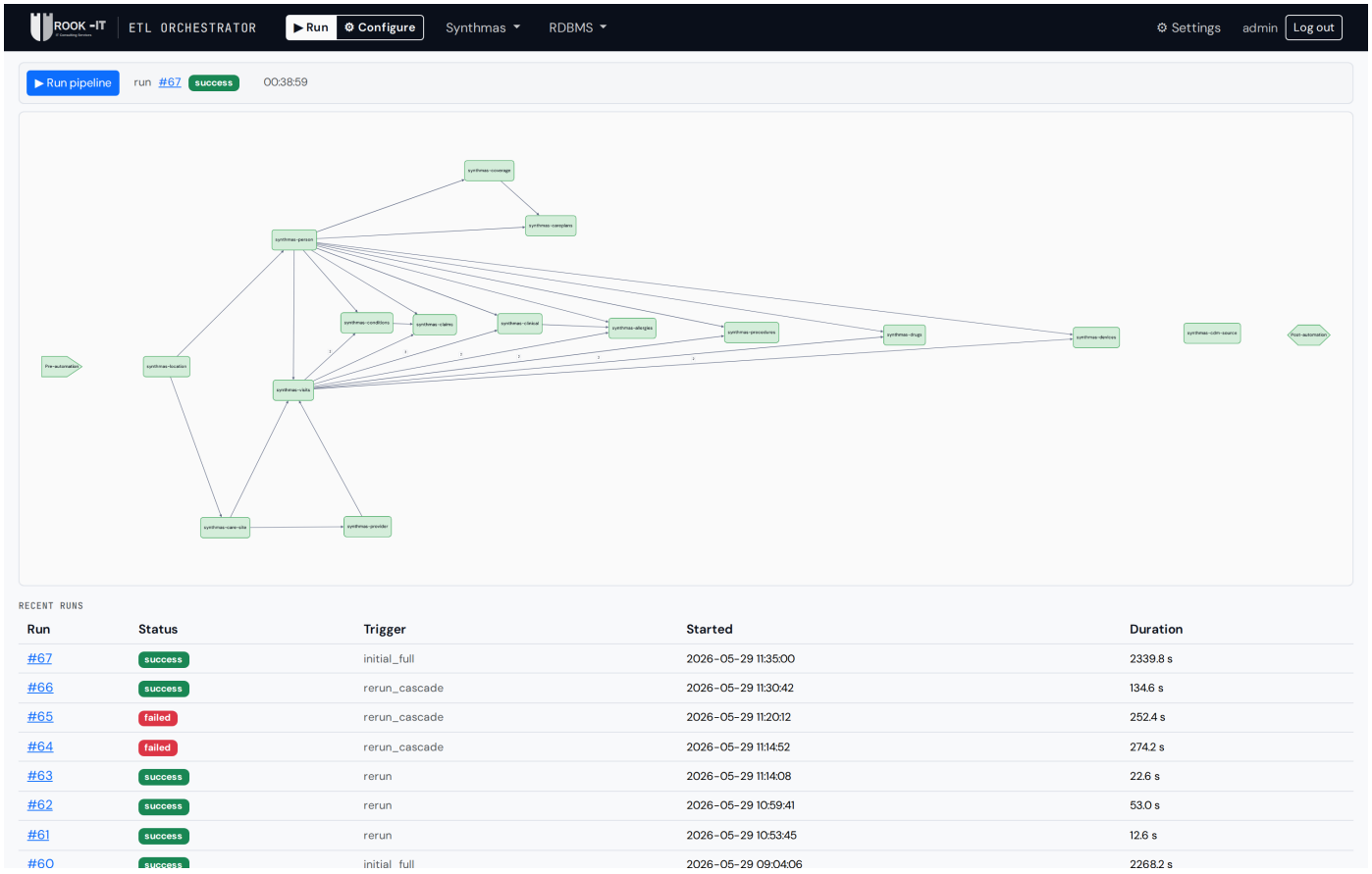
Most pipelines need more than one RiaH because a single RiaH can't express everything (multi-source fan-out into the same CDM table being the canonical example). The orchestrator auto-wires steps when:

- Step A has a `StoreIndexMap` named `X` and step B has a `GetCDMTableId(@X, …)` — that's a "consumes" edge from A to B.
- Steps A and B both write into CDM table `T` with named `StoreSequence(T_id_seq)` — they share IDs, and the dispatcher ensures A runs before B (or vice versa, whichever is consistent with other constraints).

See DAG wiring for the full mechanics.

4.6 Running a pipeline

Pipelines run **at the env level**. Pick an environment, switch to Run mode, hit ▶ **Run pipeline**.



Run mode for Synthmas / RDBMS — every node green means every step completed successfully. The Recent runs table underneath auto-updates as new runs come in and is the entry point to the full per-step history.

4.6.1 Modes

The environment page has a Configure / Run toggle in the top bar:

- **Configure** — the DAG with config affordances (upload RiaH, edit env, etc.). Greyed out when there's a run in flight to discourage changes mid-run.
- **Run** — the DAG with progress overlays: per-step bars beneath running nodes, status colors for each node, the ribbon bar at the top with overall progress, the auto-updating Recent Runs table.

4.6.2 Triggering a run

Action	Where	What it does
▶ Run pipeline	Run mode, top of page	New pipeline_run, full re-run (every step in topo order).
↻ Rerun (cascade)	Right-click any etl_map node	Recompute the cascade (this step + every shared-table writer, transitively), TRUNCATE/wipe the affected tables, re-run only the cascade members.
↻ Re-run this step	Right-click post-automation	Single-step re-run of just post-automation (observation_period + eras + preceding-visit-IDs).

You can only have **one run in flight at a time per env**. Trying to start a second one shows a clear "a run is already running" error and links to the in-flight run's detail page.

4.6.3 Live progress

Three signals while a run is in flight:

- **Ribbon bar (top)** — overall % progress, weighted by row-count estimates per etl_map step + a synthetic budget for pre/post-auto.
- **Per-step mini-bars** — beneath each running DAG node. Click any node to open the full modal.
- **Step modal** — per-table progress for etl_map steps; for pre-automation: vocab table loaded counts + an "Indexes X of 17" bar; for post-automation: era counts as each task completes.

The "Recent runs" table at the bottom of the page also auto-updates — no page reload needed. Each row links to the run-detail page with full per-step outcomes + downloadable engine logs.

4.6.4 Aborting

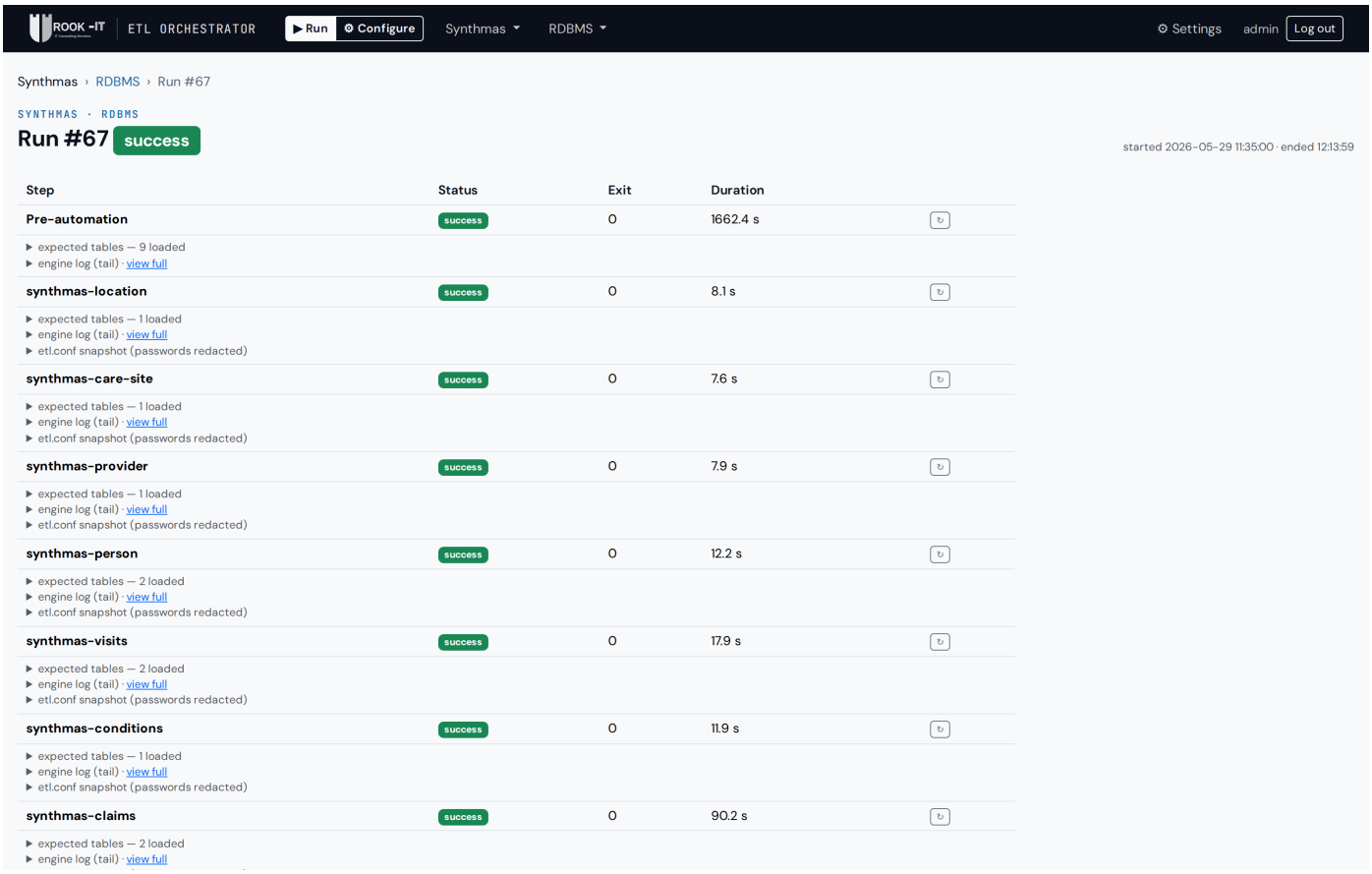
A 🛑 **Stop** button appears next to the ribbon while a run is in flight. Click it → the dispatcher stops scheduling further steps + the currently-running engine container is killed. The pipeline_run status turns `aborted`; partially-loaded data stays in the CDM (the next re-run's cleanup will handle it).

4.6.5 When a step fails

- The step turns **red**.
- The run keeps going **only** for steps that don't depend on the failed step. Anything downstream of the failure shows up as `blocked` with a `dep_check_failed` reason.
- The whole run rolls up to `partial` (some steps OK, some not) or `failed` (pre-automation died — no point running anything else).
- Click the failed step to see the engine log (live-tailed; lets you read the actual exception inline).

4.6.6 After the run

- **Run mode → Recent runs** lists every run for this env.
- **Run detail page** (click a run id) gives the full step-by-step outcome history with per-step engine logs you can download as `.log` files.



Run detail: per-step status, exit code, duration, expected-tables outcomes, and the engine log (tailed live while the step was running, downloadable after). Each step also has the rendered etl.conf snapshot for that run.

4.7 Re-running (cascade)

When you change a step's RiaH or want to re-process a step's output, the orchestrator computes the **re-run cascade**: the target step plus every sibling step that shares at least one CDM table with it (transitively). All cascade members are wiped + re-run together; nothing else is touched.

The full conceptual story is in Concepts → Re-run cascade. This page is the day-to-day how-to.

4.7.1 Two ways to trigger

From the DAG (right-click)

In Run mode, right-click any `etl_map` node → ↻ **Rerun (cascade)**.

A confirm dialog tells you exactly what's about to happen:

Re-run cascade for 'synthmas-conditions'?

Will re-run 2 step(s): • synthmas-claims • synthmas-conditions

Will wipe 2 CDM table(s): condition_occurrence, cost

Click through and the orchestrator:

1. Deletes the cascade-touched CSV files (`csv_cdm/<env>/<table>`) and the on-disk `globalSeq/<seq>` files for sequences any cascade member writes (so IDs restart at 1, not from the prior max).
2. Starts a new `pipeline_run` with trigger `rerun_cascade` and the cascade scope persisted on the row.
3. Marks every non-cascade step `skipped` with `failure_reason='rerun_skip_outside_cascade'` — they don't run, but the dispatcher trusts their prior artifacts on disk.
4. Runs the cascade members in topo order; the same shared-`globalSeq` edges that prevent races during a full run now keep the cascade internally ordered.

Pre/post automation are never cascade members (they don't write data tables, just manage CDM lifecycle). The right-click menu doesn't appear on those nodes — re-run via the full ▶ **Run pipeline** button if you need to re-init the CDM.

From the step-state modal (left-click)

Same flow, different entry. Click a node → modal opens → ↻ **Rerun (cascade)** button in the header. Same confirm dialog, same effect.

4.7.2 When the cascade is just one step

For single-writer CDM tables, the cascade is `{self}` — a true single-step re-run. In the synthmas demo that's `location` , `care-site` , `provider` , `person` , `visits` , `measurement` (clinical), `procedures` , `drugs` , `devices` , `cdm-source` . The confirm dialog will list just one member.

4.7.3 When the cascade includes others

Three multi-writer pairs in synthmas:

target	cascade	wiped tables
`clinical` ↔ `allergies`	both	`measurement` , `observation`

The pairing is symmetric: triggering either member pulls in the other. (`clinical` / `allergies` is the only shared-table pair left in the demo — `claims` now writes only `cost`, and the `careplans` step was removed; `observation_period` is produced by post-automation, not a cascade member.)

4.7.4 DAG colours during + after a cascade

While a cascade run is in flight, the DAG strip shows only the cascade members as running. The rest of the nodes stay coloured by their **effective state** — the latest run in which they actually ran, not the cascade run's "skipped" status. So after re-running `{conditions, claims}`, `clinical` / `allergies` / `procedures` etc. still show green because their data is unchanged.

The run header / ribbon (run id, timer, total progress) still ties to the literal latest `pipeline_run` — so "what cascade did I just trigger" and "what's currently in my CDM, per step" don't fight each other.

4.7.5 Re-running after an orchestrator restart

The cascade scope is persisted on `pipeline_run.cascade_step_ids_json`. If the orchestrator container is restarted mid-cascade (worker reload, host reboot), the orphan-recovery path rebuilds it and resumes correctly — no sliding outside the cascade. A per-pipeline-run file lock (`<data>/resume_locks/<id>.lock`, atomic via `O_EXCL`) keeps multiple gunicorn workers from double-dispatching the same orphan.

4.7.6 When to use the full re-run instead

Use ▶ **Run pipeline** (not cascade) when:

- The CDM should be fully reset (pre-automation wipes every pipeline-written table + the env-scoped indexes/seq state).
- You changed dimension data that everything depends on (`person`, `visits`), and a cascade through every event step would re-run most of the pipeline anyway.
- You're not sure what's affected and want a clean baseline.

4.8 Save / Load

Two single-file bundle formats — one per concept — that round-trip end-to-end:

- `.etlopipeline` — a pipeline's config + every step file. Use for backup, sharing across instances, or duplicating inside one instance.
- `.etloenv` — an environment's config (small; encryption-by-construction for password fields). Use to move an env between instances or back up the bits you don't want to re-type.

The two are intentionally orthogonal: different scopes, different sensitive surface, separate files.

4.8.1 Pipeline `.etlopipeline`

Save

Pipelines → (open pipeline) → Configure → ⤓ Save.

Split-button — the dropdown lets you choose whether to embed the vocabulary zip:

- **Save (no vocab)** — small (KB–MB). Manifest records the vocab version + sha256; import warns if no matching vocab is installed on the target.
- **Save + include vocabulary (large)** — self-contained (potentially ~1 GB). Best for cold installs.

What's inside

```
<pipeline-name>.etlopipeline       (a zip)
├── manifest.json          pipeline config + steps[] (resolved by step name)
├── blobs/<sha256>         every current step file content, sha256-deduped
└── vocab/<filename>       optional, if you ticked the include-vocab option
```

The `manifest.json`:

- **Pipeline-level**: name, description, omop_version, engine_image_ref, post-automation flags, dag_layout_json.
- `steps[]`: name, kind, display_order, new_mode, etl_conf_overrides, kind_config + `files[]` (kind, filename, blob_sha256, notes) + per-RiaH `expected_tables`, `produces`, `consumes`. Consumes use the producer step's **name**, not its id — so the bundle re-links cleanly across id spaces.
- **Schema version** for forward compatibility.

What's *excluded* (by design)

- Environments (they're per-instance; separate bundle).
- Source sets (per-pipeline but per-instance; user re-uploads).
- Runs, cycles, run history.
- License file (separate concern).

Load

Pipelines → ⤒ Import. Pick the `.etlopipeline` file.

- Always creates a **new** pipeline (name-clash gets a `(2)` suffix).
- Pre/post automation steps are seeded by the orchestrator first, then the manifest's pre/post entries update them in place (by `kind`).
- Step files de-dup via the blob store by sha256.

4.8.2 Environment `.etloenv`

Save

Pipeline → Environments → (open env) → Configure → ↓ Save.

Tiny — typically hundreds of bytes; a few KB at most.

What's inside

```
<pipeline-name>-<env-name>.etloenv     (a zip)
└── manifest.json
    ├── pipeline_ref: {name, omop_version}     compatibility check at import
    └── environment:
        - source_db_config_json + cdm_db_config_json (verbatim — see below)
        - log_level, vocab_cache_size, cdm_max_connections, worker_threads
        - use_dedup_cache, etl_conf_overrides_json
        - engine_image_ref (override)
        - engine_env_vars_json (KEY=VALUE bag for the engine container)
```

What about password security?

Password fields inside `*_db_config_json` are stored AS-IS, which is safe because they're already encrypted with the originating instance's login password (field-level encryption). The bundle file doesn't need its own encryption layer — the sensitive fields are already protected, and they stay protected on import unless the target instance shares the same login password.

The manifest carries a `note_on_passwords` field flagging:

> Encrypted-field values inside *_db_config_json are bound to the ORIGINATING instance's login password. Re-enter passwords on the target instance after import if it has a different login password.

What's *excluded*

- `csv_source_set_id` (instance-local pointer to a per-pipeline source set; user picks one on import).
- Runs, cycles, CDM state caches.

Load

Pipeline → Environments → ↑ Import. Pick the `.etloenv` file. The orchestrator:

1. Checks `pipeline_ref.omop_version` matches the target pipeline (hard error on mismatch).
2. Creates a new env under the target pipeline (name clash → `(2)` suffix).
3. Leaves `csv_source_set_id` null (you pick one in the env config page).

5. Reference

5.1 Reference

Hard facts — formats, env vars, endpoints.

- **Bundle formats** — `.etlopipeline` and `.etloenv` internal layout.
- **Environment variables** — boot-time + install knobs.
- **HTTP endpoints** — routes exposed by the web app (useful for scripting / debugging).

5.2 Bundle formats

Schema reference for the two save-load file formats. Both are zips with a `manifest.json` at the root; the orchestrator versions the manifest so forward-compatible loaders can be added later.

5.2.1 `.etlopipeline`

```
<pipeline-name>.etlopipeline
├── manifest.json
├── blobs/<sha256>          # one per current StepFile, sha256-deduped
└── vocab/<filename>        # optional — only when exported with "include vocab"
```

`manifest.json` (pipeline)

- **Pipeline-level fields**: name, description, omop_version, engine_image_ref, post-automation flags, dag_layout_json.
- `steps[]` : name, kind, display_order, new_mode, etl_conf_overrides, kind_config + `files[]` . Each RiaH file carries `expected_tables` , `produces` (`{kind, name}`), and `consumes` (`{kind, name, producer_step: <step name>}`).
- `vocab` : `{vocab_version, blob_sha256, filename}` or `null` .

Wiring is resolved by step **name**, not id, so the bundle re-links cleanly across id spaces.

Import behaviour

- Name clash → `(2)` suffix on the new pipeline.
- Pre/post automation steps are seeded by `ensure_automation_steps()` first, then the manifest's pre/post entries update them in place by `kind` .
- Step files de-dup via the blob store by sha256 (existing blob = reuse).
- Wiring is materialised in a second pass after every step exists.

Excluded (by design)

- Environments, source sets, runs, cycles, licenses.

5.2.2 `.etloenv`

```
<pipeline>-<env>.etloenv
└── manifest.json
```

`manifest.json` (environment)

- `pipeline_ref` : `{name, omop_version}` — compatibility check on import.
- `environment` : name, both `*_db_config_json` blobs (verbatim — see encryption note below), log_level, vocab_cache_size, cdm_max_connections, worker_threads, use_dedup_cache, etl_conf_overrides_json, engine_image_ref override, engine_env_vars_json (KEY=VALUE bag for the engine container).

Encryption-by-construction

Password fields inside `*_db_config_json` pass through as the orchestrator stored them. Already-encrypted values stay encrypted in the bundle — the encryption is bound to the originating instance's login password, so the bundle file itself doesn't need an extra encryption layer.

If the target instance has a different login password, you'll need to re-enter the passwords after import. The `note_on_passwords` field in the manifest is informational.

Import behaviour

- OMOP version mismatch → hard error before creating the env.
- Name clash → `(2)` suffix.
- `csv_source_set_id` intentionally NOT restored — pick one in the env config after import.

Excluded (by design)

- Run history, cycles, CDM state caches.
- `csv_source_set_id` (instance-local pointer).

5.3 Environment variables

The container honours a small set of bootstrap env vars. All have safe defaults — you only need to touch them when running the install script.

5.3.1 Boot-time

Variable	Default	Purpose
`ORCHESTRATOR_DATA_DIR`	`/var/lib/orchestrator`	Root for the SQLite DB, blobs, runs, logs, csv_cdm, indexes. Persistent — bind-mount this from the host.
`ORCHESTRATOR_LISTEN_PORT`	`8000`	Port gunicorn binds inside the container. The installer maps this to the same host port (override with the same name when running `install-etl-orchestrator.sh`).
`ORCHESTRATOR_DB_PATH`	`<DATA_DIR>/orchestrator.sqlite`	Override only if you need to point at a different DB file.
`ORCHESTRATOR_CONTAINER_SOCKET`	`(auto)`	Path to the docker / podman socket. Probed in order: `/var/run/docker.sock`, `/run/podman/podman.sock`, `$XDG_RUNTIME_DIR/podman/podman.sock`. Override only if probing finds nothing.

5.3.2 Installer-only (host-side)

These are read by `install-etl-orchestrator.sh` and persisted to `bin/etlo.config` for the lifecycle script:

Variable	Default	Purpose
`ORCHESTRATOR_DATA_DIR`	`/var/lib/orchestrator`	Host path to bind-mount as `/var/lib/orchestrator` inside the container.
`ORCHESTRATOR_LISTEN_PORT`	`8000`	Host port to publish.
`ORCHESTRATOR_CONTAINER_NAME`	`etl-orchestrator`	Container name. Override when running multiple installs on the same host.

5.4 Endpoints

(Page under construction — content forthcoming.)

6. Concepts

6.1 Concepts

Background pages — the *why* behind the *how*.

- **Re-run cascade** — the shared-table semantics that scope selective re-runs.
- **DAG wiring** — how `GetCDMTableId(@table)` lookups and `StoreSequence(…)` writers become edges in the pipeline DAG.
- **CSV / PostgreSQL parity** — why the CSV target produces the same data as the PostgreSQL target, what engine version you need, and the three asymmetries that remain (vocab loading, eras, cascade cleanup).

6.2 Re-run cascade — shared-table semantics

6.2.1 The problem

You've changed one step's RiaH (say `conditions`) and want to re-load `condition_occurrence` without re-doing everything. Two naive answers are both wrong:

- **"Re-run only `conditions`."** Wrong. `conditions` and `claims` both write `condition_occurrence` (claims appends via a named `StoreSequence`). If we re-run only `conditions`, its rows replace the original 1..N range, but `claims`' rows from the prior run still sit at N+1..M referencing the old IDs. The CDM is left silently inconsistent.
- **"Re-run `conditions` + everything topologically after it."** Wrong — too greedy. It re-runs `clinical`, `allergies`, `procedures`, `drugs`, ... — anything appearing later in topo order, even if they don't share any CDM table with `conditions` and thus aren't affected.

6.2.2 The right answer

The re-run **cascade** is the connected component of steps you reach from the target by walking the "shares a written CDM table with" relation, transitively. Concretely:

```
cascade(N) = { N } ∪ { every step S that shares ≥ 1 written CDM table with
                       some step already in cascade(N) }
```

Why this is exactly right: the engine's per-table cleanup is `DELETE FROM T WHERE pk(T) >= captured_seq_before_N`, which wipes everything `N` wrote *and* everything any later step wrote to the same table (their IDs are higher than `N`'s captured seq). So you can't selectively keep one writer's rows. The downstream set the wipe forces to re-run is, by construction, every other writer of any of `N`'s tables — and transitively their tables' other writers — i.e. the cascade.

For tables with one writer (the common case), the cascade is `{N}` — a true single-step re-run.

For shared tables, the cascade always contains every writer. In synthmas:

trigger	cascade	tables wiped
`clinical` ↔ `allergies`	both	`measurement`, `observation`
any single-writer step	self	one table

(`clinical` and `allergies` both write `observation` — the one shared-table pair left in the demo. `conditions` / `claims` no longer share a table — `claims` now writes only `cost` — and the `careplans` step was removed.)

Pre/post automation never enter the cascade — they don't write source-driven data tables. Note post-automation now *does* produce `observation_period` (one per person, both backends) and, on RDBMS, the eras + preceding-visit-IDs.

6.2.3 CSV CDM cleanup, concretely

The general DELETE-pivot rule:

```
pivot = max(captured_seq.next_id across NON-cascade writers of T)
      = 0   when every writer of T is in the cascade (the common case)
DELETE FROM T WHERE pk(T) >= pivot
```

In CSV mode, with the cascade being closed (every writer is included by construction), `pivot = 0` always → `DELETE FROM T` = delete the whole CSV file. The orchestrator's `csv_cdm_cleanup` does exactly that:

1. `/csv_cdm/<env>/<table>` — removed for every cascade-touched table.
2. `/indexes/<env>/globalSeq/<seq>` — removed for every named sequence any cascade member produces (so AutoGenId restarts at 1, not from the prior max).
3. `/indexes/<env>/<table>/` — removed for every `StoreIndexMap` a cascade member produces (the engine rewrites it during the run).

Index dirs produced by steps *outside* the cascade are left intact — their non-cascade consumers (which we are NOT re-running) still depend on them.

6.2.4 What survives across runs

`shared_indexes` is **env-scoped** (`/indexes/<env>/`), not per-run. Cascade re-runs need this — the cascade target's upstream isn't re-run, but the target still has to read the upstream's `StoreIndexMap` on disk. Full re-runs reset cleanly because `pre_automation` wipes the env-scoped indexes dir as part of its CDM-lifecycle work.

6.2.5 Resume semantics

`pipeline_run.cascade_step_ids_json` persists the cascade scope. If the orchestrator container is restarted mid-cascade (worker reload, host reboot), the resume path:

1. Acquires `<data>/resume_locks/<pipeline_run_id>.lock` via `O_EXCL` — only one gunicorn worker can resume a given orphan. Stale-pid sweep handles the case where the prior owner died without releasing.
2. Reads `cascade_step_ids_json` and re-passes it to `_dispatch` so the skip-outside-cascade branch keeps firing for the rest of the run.
3. The cleanup phase doesn't re-run on resume (it ran in the original kick-off; resume just continues the topo loop).

Without (2) the resumed dispatcher would treat every step with no prior step_run as a normal dispatch target and slide outside the cascade. This is the bug behind the "why did clinical run when I asked for a conditions cascade?" question that drove the persistence design.

6.2.6 Single-flight, abort, and the cascade

The orchestrator's single-flight guard ("one `pipeline_run` in flight globally") applies to cascades too: a cascade re-run is just another `pipeline_run` , and you can't trigger another while one is running. Abort works the same way — `pipeline_run.abort_requested` propagates across workers.

6.3 DAG wiring

The pipeline DAG you see in Configure mode isn't drawn by hand. The orchestrator derives it from what's *in* the RiaH files, with a small amount of OMOP-aware enrichment on top. This page is about **how** it infers those edges and **why** you can trust the result.

6.3.1 What the engine actually needs

Most multi-RiaH pipelines share state between steps. The engine has exactly two cross-step state mechanisms:

- `StoreSequence(<name>)` — write a named global sequence value to disk. Subsequent steps continue from there via `AutoGenId(<name>)`. Used when two RiaHs both append rows to the same CDM table (e.g. `claims` and `conditions` both writing into `condition_occurrence` — they share `condition_occurrence_id_seq`).
- `StoreIndexMap(<source_key…> → <cdm_key…>)` — write a per-key lookup file. Subsequent steps read it via `GetCDMTableId(@<table>, …)` to resolve a source-side reference into the CDM ID an earlier step assigned.

Together, those two are how the engine's worker threads stay parallel-safe across steps: writes are local to one step, reads are materialized as files between steps.

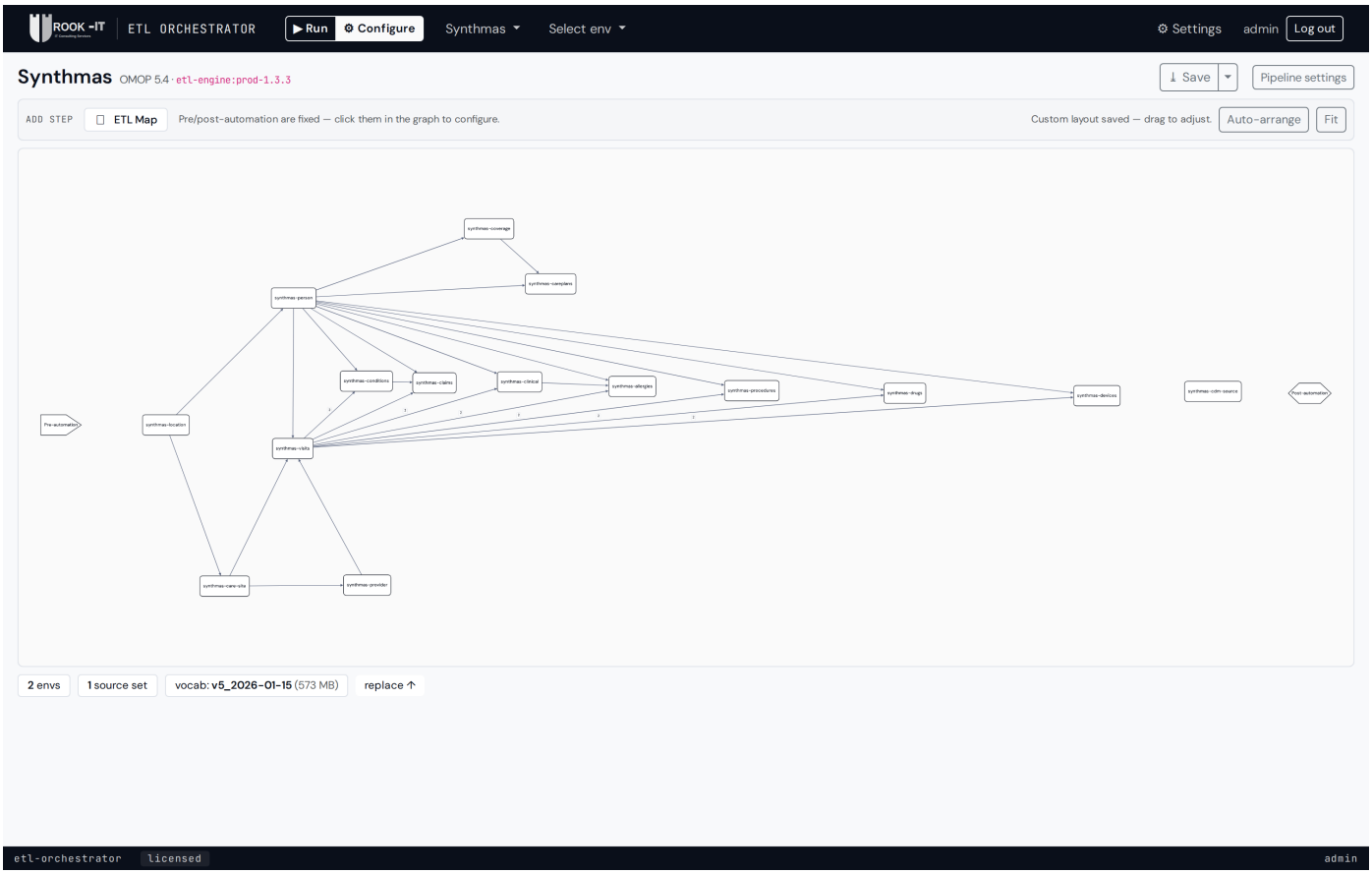
6.3.2 How the orchestrator parses this

On every RiaH upload, the orchestrator parses the file and records:

- **Produces** — the names of every `StoreSequence` and `StoreIndexMap` rule the RiaH declares.
- **Consumes** — every `GetCDMTableId(@X, …)` and `AutoGenId(<name>)` that references a name some other step produces.

These are stored against the step's RiaH version, so updating a RiaH re-derives the edges immediately.

6.3.3 How the edges get drawn



The Synthmas DAG — every edge here was derived from parsing the RiaHs, never declared by hand. `synthmas-person` produces `index/person`; `synthmas-conditions`, `-claims`, `-allergies`, `-immunizations`, `-medications` all consume it (hence the fan-out from the centre-left).

For each pair of steps A, B:

- If A produces `X` and B consumes `X` → solid edge **A → B**, labeled with the artifact name. That's the **artifact wiring** layer.
- If A and B both write into CDM table `T` (per their expected-tables lists) → they're declared *related* (no edge — they collaborate via shared StoreSequence — but they share a cascade, see Cascade).

The orchestrator additionally surfaces:

- **FK dependencies** between the tables A writes and tables B writes, derived from the OMOP CDM schema for the pipeline's version. These render as **dashed edges** in the configure DAG — they're informational ("B references rows A creates via foreign key"), not execution-gating.

6.3.4 What you don't need to specify

- Step order. The dispatcher topo-sorts the DAG before each run.
- Edge directions. They come straight from "X is produced here, consumed there."
- Shared-table coordination. The auto-detected shared-writer relationship is what powers cascade re-runs.

If two of your steps SHOULD share state but the orchestrator hasn't drawn an edge, the RiaH side of the wiring is the place to look — chances are the producer is using an unnamed `StoreSequence` or the consumer's `GetCDMTableId` is referencing a literal table instead of `@<storedIndexName>`.

6.3.5 What the orchestrator does NOT auto-infer

- **External data dependencies**. If your step's mapping refers to a Usagi or CSVMap file, those aren't artifact-wiring inputs from another step — they're per-step lookup files you upload alongside the RiaH.
- **Vocabulary**. Vocab is pipeline-scoped, loaded once per env by pre-automation. There's no per-step "I depend on the vocab" edge — every step does, implicitly.

6.4 CSV / PostgreSQL parity

The CSV CDM target and the PostgreSQL CDM target produce the **same data** when you run the same pipeline against both with the same input. Not "approximately the same" — every row in every CDM table matches by value, in the same order, with the same `rejected` counts. The difference is the destination, not the data.

This page exists because that wasn't always true. CSV used to be excluded from the orchestrator's supported-backends list (the original argument was "no global state to manage" — the engine wrote per-table files lazily, with no DDL phase and no canonical-header guarantee). Engine 1.4.0 and 1.4.1 closed that gap; CSV is now first-class.

6.4.1 Why it works now

Four engine fixes, in order.

Engine commit	What it does	Why it matters
`a43b83f` (1.4.0)	`create omop cdm: once` now writes each CSV CDM file with the **canonical OMOP header** on day-zero.	Multi-writer tables (e.g. `cost`, written by both `synthmas-claims` and `synthmas-cost-drugs`) see the full column set regardless of write order. No more "field X is missing in table" on first-writer-locked schemas.
`f928368` (1.4.0)	`clean omop data: true` now rewrites each CSV file to its canonical header on day-N (SQL TRUNCATE semantics: schema preserved, body empty).	Day-N runs preserve the canonical schema instead of falling back to lazy first-write. The orchestrator can pin one cleanup config for both CSV and RDBMS.
`4519128` + `8131819` (1.4.1)	`CSVTable::loadReadTable` skips virtual fields in header validation.	Virtual fields declared in RiaH table comments (e.g. `field: visit_occurrence_source_value` used only as a `StoreIndexMap` key) don't trip the parse-time column-presence check against the canonical OMOP header.
`213521d` (1.4.1)	`CSVTable::loadWriteTable` opens the existing file with `out | app` instead of `out | ate`, so the canonical header laid by `create omop cdm` / `clean omop data` survives the first writer's append.	The original `out | ate` open silently truncated the file (per the C++ standard, `out` alone truncates). Without this fix, the first writer landed a data row at offset 0 and the second writer's read-side check then tripped.

The orchestrator side was simplified at the same time: the CSV pre-automation special-casing went away (`dispatcher.py` no longer has a CSV branch in `cleanup_scope`, no `csv_cdm_cleanup` call, no orchestrator-side short-circuit in `_run_step`). Both backends now emit the same `etl.conf` shape — the only difference is the `cdm database.driver` string.

6.4.2 How to verify

The synthmas demo's parity comes out at **2,015,051 rows across 16 CDM tables**, identical CSV vs PG. Six PG runs back-to-back also produce identical output to each other (the older "PG write path is non-deterministic" report is now resolved).

The verification snippet is in **Run on PostgreSQL → Confirm CSV / PostgreSQL parity** — pass it the two `pipeline_run` ids (one CSV, one PG) and it prints a per-table delta column that should be `+0` on every row.

6.4.3 Pin engine 1.4.1+

The orchestrator's settings page lets you set a per-pipeline engine image. For CSV envs, pin `rook-it.com/etl-engine:1.4.1` or newer. Earlier engines have known correctness gaps on multi-writer CDM tables — the pipeline will abort with "field … is missing in table" on `synthmas-cost-drugs` (and similar steps in your own pipelines).

For RDBMS-only envs, the version pin matters less for correctness, but keeping CSV and RDBMS envs on the same engine image makes parity testing trivial — same image, same pipeline, both backends, identical output.

6.4.4 What stays asymmetric

Three things still differ between the backends. None of them affect CDM table row counts; they're all about what happens around the data.

- **Vocabulary loading** on CSV is orchestrator-side (the orchestrator extracts the Athena vocab zip into `csv_cdm_dir` before the engine spawns). The engine's `populate vocabulary` path INSERTs into vocab tables via SOCI and is RDBMS-only. End-state is the same — pre-populated CONCEPT.csv et al. — just a different code path.
- **Era + preceding-visit-IDs post-automation** is RDBMS-only. The orchestrator forces those flags off on CSV regardless of the pipeline's toggle (`condition_era` , `drug_era` , `dose_era` , and `visit_occurrence.preceding_visit_occurrence_id` stay empty / `NULL`).
- **Cascade re-run cleanup** on CSV relies on the canonical header pre-automation already left in place — there's no CSV equivalent of the RDBMS `rdbms_cdm_cleanup` per-table DELETE-pivot. For a true re-clean of cascade tables on CSV, run the full pipeline instead of the cascade.

These are documented in the etl.conf template reference and in the CSV-side-of-the-house design notes; they're stable, intentional asymmetries — not parity gaps.