Senior Data Engineer DML Capital Group Take-Home — Factoring Fund ETL MVP

We purchase/advance against future insurance commission receivables. We ingest bank exports and insurer/broker statements to: (a) audit collections, (b) compute the borrowing base (how much we can safely advance), and (c) monitor delinquencies to ensure obligations are met.

Timebox: 2–6 hours. Focus on an end-to-end MVP. Clear trade-offs > feature breadth. If you cannot complete a certain part due to time constraints , please think how you would implement it and explain during the interview.

The business problem

We need an automated daily pipeline that:

1. Ingests raw data from bank transaction exports and expected commission schedules.

2. Reconciles expected vs. received cash.

3. Computes an **as-of-date borrowing base** per facility using rules below.

4. Produces exception reports and metrics we can review each morning.

You’ll work from the provided synthetic CSVs (see `data/`), assume CAD currency, and implement an MVP that would be productionizable with reasonable effort.

Data you'll use (synthetic)

`facilities.csv`

- `facility_id` (PK), `counterparty_name`, `max_advance_rate` (0–1), `reserve_rate` (0–1), `concentration_limit_pct` (0–1), `delinquency_cutoff_days` (int), `current_outstanding` (amount), `currency`.

`policy_commissions.csv` (expected receivables)

- `policy_id` (PK composite with due_date), `facility_id` (FK), `insurer`, `insured_name`, `policy_number`, `commission_due_date` (date), `commission_amount` (amount), `written_date` (date), `status` (active/cancelled), `currency`, `expected_bank_memo` (string to help matching bank memos).

`bank_transactions.csv` (actual collections + noise)

- `txn_id`, `date`, `amount` (positive = deposit, negative = withdrawal), `currency`, `description` (free-text memo), `account_id`.

> Tip: Transactions for a policy often include the insurer name and policy number in `description` (e.g., `DEP Aviva COMM POL1007`). There are also late, short-pay, missing, and unrelated/noisy deposits.

What to build

A) System design (1–2 pages in your repo)

Describe an MVP architecture **and** the path to scale:

- Layering (raw/bronze → cleaned/silver → marts/gold).
- Storage and compute choices (you may use Postgres from `docker-compose.yml` or DuckDB). Where/how you'd use PySpark vs SQL.
- Idempotency & incremental logic (e.g., watermarking by date, upserts, dedupe).
- Data quality checks (e.g., nulls, schema drift, amounts, DPD rules) and how failures alert/stop the run.
- Backfills, reprocessing, and lineage.
- Security considerations for bank data (PII, secrets management, audit log).

B) Code (Python, PySpark, SQL) + Docker

Build an end-to-end pipeline that:

1. **Ingests** the CSVs into a raw layer.

2. **Cleans/standardizes** types, dates, and normalizes memos for fuzzy matching.

3. **Reconciles** expected commissions to bank deposits:

  * Match by policy number/insurer in memo; handle late/short-pay; allow partial matches.

* Output `reconciled_collections` with: policy_id, expected_date, expected_amount, received_date (nullable), received_amount (nullable), shortfall, **DPD** (days past due), and a `match_confidence`.

4. **Computes the borrowing base** per facility for an as-of date *(default: latest txn date or today)*:

- **Eligible receivables** = expected but uncollected items **not** over the facility's `delinquency_cutoff_days`.
- Apply **concentration limit**: sum eligible by insurer; cap each insurer at `concentration_limit_pct` of total eligible before advance.
- **Advance** = `eligible_sum * max_advance_rate`.
- **Reserve** = `eligible_sum * reserve_rate`.
- **Borrowing base** = `Advance − Reserve`.
- **Headroom** = `Borrowing base − current_outstanding`.
- Output a `borrowing_base_summary` per facility with the intermediate columns so the math is auditable.

5. **Persists outputs** to your warehouse (Postgres or DuckDB) **and** writes CSV extracts to `outputs/`:

- `borrowing_base_summary.csv`
- `exceptions.csv` (e.g., unmatched, >X days late, short-paid >Y%).

6. **Scheduling**: add a small orchestrator (e.g., **Prefect 2** flow, or Dagster/Airflow if you prefer) and schedule it for daily 07:00 **America/Toronto**. Include a manual backfill option for a date range.

7. **Observability**: basic logging; emit run metadata and record data row counts.

C) Presentation (5–10 slides or README section)

- Explain the business problem in your own words.
- Call out assumptions & trade-offs for the MVP.
- Walk through the data model and pipeline.
- Summarize findings (e.g., delinquency distribution, headroom by facility) and **risks/concerns**.
- Propose next steps to productionize and scale.

Constraints & guidance

- **Must use**: Python, **PySpark** (local mode fine), SQL, Docker. Use the provided `Dockerfile` as a starting point or your own.
- **Scheduler**: Prefect 2 recommended for speed; cron acceptable if justified. Airflow/Dagster = bonus if you keep it simple.
- Keep it runnable via `docker compose up` **or** a single make/poetry command. Include seed `.env.example` if needed.

Acceptance criteria (the MVP is "done" when):

- ✅ Pipeline runs end-to-end inside Docker and produces the three outputs (`reconciled_collections` table, `borrowing_base_summary.csv`, `exceptions.csv`).
- ✅ PySpark is used materially in at least one transformation step (not just imported).
- ✅ SQL is used for at least one business-logic aggregation (e.g., borrowing base calc or exception rollups).
- ✅ A scheduler is set to run daily at 07:00 Toronto time and supports a manual backfill.
- ✅ Clear README with setup, commands, and architecture notes; plus a short presentation section/slides.

Stretch goals (optional)

- Great Expectations (or similar) for data quality; surface failures.
- dbt or SQL model files for marts.
- Incremental logic (watermark on dates) and upserts to warehouse.
- Basic CI (lint/tests) and pre-commit hooks.
- Lightweight dashboard (e.g., Streamlit) to visualize headroom & delinquencies.

Evaluation rubric (100 points)

- **Business understanding & assumptions (20)** — Can you restate the problem and choose reasonable rules?
- **Correctness & reproducibility (20)** — Deterministic outputs; idempotent runs; clear instructions.
- **PySpark & SQL proficiency (20)** — Appropriate use of Spark APIs and SQL joins/aggregations.
- **Design & scalability (15)** — Layering, partitioning, incremental strategy, backfills.
- **Scheduling & observability (10)** — A real schedule + usable logs/metrics; simple alerts ok.
- **Data quality & testing (10)** — Checks for schema/amounts/dates; a few unit/data tests.
- **Communication (5)** — Clear README/presentation, trade-offs called out.

Submission

- A public Git repo (or zipped archive) with code, README, and document/slides for presentation.
- Include exact commands to build and run (e.g., `docker compose up --build`, or `make run`).
- If anything is unclear, state your assumptions in the README and proceed.

Good luck — excited to see your approach!