Case Study 1 — Automated Data-Quality Framework (HEXA VPN)

Role: Data Analyst · **Company:** Hazel Mobile · **Product:** HEXA VPN app

Stack: PostgreSQL (advanced SQL) · Python · data-quality engineering

Deliverables: automated DQ check suite · before/after impact report · methodology guide · architecture diagram

> *Figures in this case study and its documents are illustrative / synthetic for portfolio use — they show the method, not real production data.*

The Problem

HEXA VPN's dashboards reported a healthy connection **success rate** — but that headline counted *every* session that merely *reported* success. In reality, many of those weren't genuine wins: connections that dropped within seconds ("flaps"), sessions instantly replaced by an app bug, automated/bot traffic, corrupt records, and users who connected once and never returned. Leadership was making decisions on a number that **overstated true product health**. The company needed a way to measure the *real* success rate — and to know exactly *why* it differed.

What I Built

I designed an **automated, read-only data-quality framework** in SQL that scores the telemetry and **reclassifies "fake" successes to failures** — without ever deleting an event. It runs as three tiers of checks:

Tier	Question it answers	Examples
Row-level (#1–#15)	Is each record clean, valid, and plausible?	timestamp validity, plausible session length, valid server IP, self-replace-bug, plus the data-integrity checks below
Behavioural (B1–B10)	Is each account behaving like a real user?	inactivity churn, bot/burst detection, one-and-done, country-hopping
Journey (T1–T3)	Did the user get (and keep) a good session?	flap (<30s), activation, next-day return, churn-risk

The row-level tier runs **fifteen checks (#1–#15)** in one modular pass — from basic record hygiene (valid timestamps, plausible durations, well-formed server IPs, the self-replace-bug guard) through to data-integrity and degraded-experience signals:

- **#12 · Implausibly long sessions** — sessions running longer than three days (a duration-metering integrity signal).
- **#13 · Future-dated timestamps** — events that claim to have happened in the future (device clock-skew / impossible records).

- **#14 · Free-user entitlement anomalies** — free accounts carrying a day or more of accumulated connection time (a metering / granted-balance check).
- **#15 · Fallback-route connections** — sessions that only connected after switching to a backup route, i.e. a *degraded* connection rather than a clean first-attempt success.

On top of the checks sits a **before/after impact model**: it computes the success rate as reported vs. the true rate after reclassification, splits the affected users into *one-and-done* vs. *retained*, and attributes the drop to individual checks. Attribution runs in **priority order**, so each check is credited only with the *unique* records it catches and the pieces always sum to the total.

The Techniques

The framework is a showcase of production-grade SQL:

- **CTEs** to stage the logic in readable layers (scan → flags → aggregates).
- **Window functions** — `LAG()/LEAD() OVER (PARTITION BY … ORDER BY …)` to detect telemetry gaps and the self-replace bug (a session replaced, then the same account reconnecting within 60 seconds).
- **`FILTER` aggregates** to compute before/after success in a single pass.
- **`CASE` tiering** for per-check status (OK / FLAGGED / ALERT) and tier-aware thresholds.
- A built-in **self-check** that re-derives key totals a second, independent way and reports **PASS/FAIL**, so the numbers can be trusted.

Everything is **SELECT-only** and version-scoped, so it is safe to run against a live database.

The Impact

The framework surfaced a large gap between the **reported** and the **true** success rate *(illustrative: ~92% reported → ~68% true)* — and, just as importantly, explained it:

- **The drop was churn, not corruption.** Row-level data quality was strong; the erosion came almost entirely from **inactivity churn** (users who connected but never returned).
- **The targeted checks added precision, not noise.** The self-replace and bot/burst checks each flagged a small, well-targeted set — confirming there were *no* bots, only a few borderline accounts — without punishing real users.
- **Most "lost" users had barely engaged** (the large majority were one-and-done), so the business could focus on the genuinely *retained* users worth acting on.

What's in this case study

- **Architecture diagram** — how the framework flows from telemetry → checks → the reclassification model → the reports.
- **DQ Impact Report (PDF)** — the headline before/after tables and per-check attribution.
- **DQ Checks Explained (PDF)** — a plain-English guide to what every check does and why it matters.
- **SQL Highlights (PDF + .sql)** — a curated, syntax-highlighted excerpt of the actual query craft.

Skills demonstrated: advanced SQL (CTEs · window functions · `FILTER` aggregates · `CASE` · subqueries) · data-quality engineering · data validation / self-checks · behavioural & churn analysis · analytical storytelling · stakeholder reporting.