Automated Customer Data Ingestion Flow Proposal

Goal

Safely ingest mapped CSV exports from third-party systems into our production MySQL, preserving **Customer Vehicle Invoice Line Item** relationships, with auditability, rerun safety, and guaranteed file integrity.

Proposal:

Since Extract and Transform steps in ETL process is handled by Third Party, I would propose a focused ingestion flow, an automated **Cloud Run Job** that provides the **speed, safety, and control** we need without the overhead of an ETL platform.

Our current ingestion scope is **narrow and well-defined**:

- **Source**: Partner delivers pre-mapped, clean CSV files.
- **Transform**: All transformation is handled upstream by the partner.
- **Our responsibility**: **Load** data into MySQL while preserving parent–child relationships, ensuring idempotency, and maintaining an audit trail.

Using a custom Cloud Run Job instead of an ETL platform at this stage gives us:

1. **Full control over load behavior**
    - We manage crosswalk mappings, parentchild load order, and idempotent upserts — capabilities many ETL tools can't model cleanly without custom code.
2. **Performance optimized for MySQL**
    - Bulk loading via LOAD DATA LOCAL INFILE is significantly faster than generic ETL tool connectors.
3. **Lower complexity and cost**
    - No licensing fees, row-based pricing, or vendor lock-in.
    - Minimal moving parts: GCS Cloud Run Job MySQL.
4. **Fit for our current scale**
    - Single partner feed, predictable schedule, well-structured files.
    - Lightweight monitoring (Cloud Logging + Teams/email) covers our operational needs.

We will re-evaluate an ETL tool when:

- We onboard **many** partners or source types beyond CSV.
- Non-engineering users need to run, monitor, or configure pipelines through a UI.
- Governance and audit requirements expand beyond what our load_ledger and logging provide.

So, For our current use case, a focused, serverless **Cloud Run Job** provides the **speed, safety, and control** we need without the overhead of an ETL platform.

Approach:

Actors

- **3rd Party Partner** (data provider)
- **ETL Platform** (GCS + Eventarc + Cloud Run Dispatcher + Cloud Run Job)
- **Core DB** (MySQL on VM)
- **Ops** (monitoring & support)

Data contract

- One CSV per entity; columns per agreed specifications.
- Each child CSV carries its parent **source keys**.
- **manifest.json** per load contains:

```
{
  "load_id": "2025-08-22_partnerA",
  "files": [
    {"name": "customers.csv", "rows": 1234, "sha256": "d6f1a97..."},
    {"name": "vehicles.csv", "rows": 2311, "sha256": "b781f89..."}
  ]
}
```

End-to-end ingestion steps

1. **Partner upload**: CSVs + manifest.json to Google Cloud Storage (imports/{partner}/{load_id}/).
    a. Either partner uploads directly or we can let our sales team upload files from partner
2. **Auto-trigger**: **Eventarc** detects manifest.json creation and invokes **Dispatcher service**.
3. **Start Job**: Dispatcher calls **Cloud Run Job** with bucket + prefix args.
4. **Download & validate**:
    - Download all CSVs from the load folder to /tmp in the Job.
    - Verify SHA-256 checksum for each file matches the manifest.
    - Optionally check schema and do light referential pre-checks.
5. **Stage load**:
    - Bulk-load each CSV to staging_* tables using LOAD DATA LOCAL INFILE.
    - Keep load_id column for tracking.
6. **Build crosswalks**:
    - Insert missing mappings source_* internal_id using deterministic IDs.
    - This make imports **idempotent**, preserve **relationships**, and let you safely **rerun** loads without dupes
7. **Upsert to prod**:
    - Transactional upserts in dependency order: customers vehicles invoices line_items.
8. **Reconciliation**:
    - Compare row counts: manifest staging prod.
    - Write a load_ledger entry (load_id, counts, checksums, status).
9. **Notify** (optional):
    - Send summary to Teams/Email: inserted, updated, rejected counts.
10. **Audit retention**:
    - Keep raw files in GCS for 90 days.
    - Keep staging rows with load_id for traceability.

Non-functional requirements

- **Idempotent**: re-running the same load_id produces the same results, no duplicates.
- **Scalable**: Cloud Run Job can be scaled up/down; parallel loads supported.
- **Secure**:
    - VPC connector to private MySQL.
    - Least-privilege IAM.
    - Signed URLs for vendor uploads (optional).
- **Observable**: Cloud Logging, Error Reporting; (optional) dashboard from load_ledger.

Timeline Estimations:

Week 1-4 – Finalize data contract, set up GCS bucket, VPC connector, deploy Job & Dispatcher (baseline).

Week 5 – Dry-run with partner sample, add reconciliation output, performance tuning.

Week 6 – Add rejects tables, Teams/email summary, final docs & handover.

Additional Implementation Notes:

Why We need a manifest.json

manifest.json is the single source of truth for each load. It tells our job *what to ingest*, *how to verify it*, and *how to process it safely and idempotently*. Without it, we risk partial loads, silent corruption, and inconsistent reruns. manifest.json turns a pile of CSVs into a verifiable, replayable, and auditable *pack age*. It's the cornerstone of safe, automated ingestion.

What it contains (minimum)

```
{
  "load_id": "2025-08-22_partnerA",
  "source_system": "PartnerA",
  "files": [
    {"name":"customers.csv","rows":1234,"sha256":"<hex>"},
    {"name":"vehicles.csv","rows":2311,"sha256":"<hex>"},
    {"name":"invoices.csv","rows":987,"sha256":"<hex>"},
    {"name":"line_items.csv","rows":4021,"sha256":"<hex>"}
  ],
  "created_at": "2025-08-22T09:15:00Z",
  "schema_version": "v1"
}
```

Why it's required

1. **Idempotency & Replay Safety**
    - load_id uniquely identifies a drop. Rerunning the same load_id is deterministic (no dupes) and auditable.
2. **Integrity Guarantees**
    - Perfile SHA256 prevents corrupted/partial uploads and tampering before any DB write (failfast).
3. **Completeness Check**
    - Declared file list + expected row counts ensures we load *all* entities (no missing child files).
4. **Contract Validation**
    - schema_version ties files to an agreed column spec. If the partner changes columns, we detect and stop cleanly.
5. **Deterministic Orchestration**
    - The job uses the manifest to stage and load in the correct parentchild order (customers vehicles invoices line_items).
6. **Audit & Traceability**
    - Manifest is logged/stored with the run. Months later we can prove *exactly which files and checksums* were ingested for a given load_id.
7. **Operational Automation**
    - Presence of manifest.json is the trigger to start the job (Eventarc). No guessing, no halfready folders.
8. **Partner Accountability**
    - Clear handoff: the partner is responsible for producing a complete, verified package (files + checksums + counts) before ingestion.

Validation using the manifest.json

- Confirms every listed file exists in the prefix and no extra unexpected files are processed.
- Recomputes and matches SHA256 for each file.
- Optionally compares row counts (manifest vs staging) postload and records any variance.
- Logs the entire manifest alongside run metadata into a load_ledger.

Error handling policy

- If any checksum, file presence, or schema check fails: abort the run, mark load_id as failed in load_ledger, and notify with a clear reason. Nothing touches prod.

Why checksums (SHA-256) matter

Checksums in manifest.json are a **critical safeguard** in the ingestion pipeline:

1. **Corruption detection** – Ensures no file was damaged or truncated during upload or transfer.
2. **Version guarantee** – Confirms the ingested file is exactly the version approved by the partner, no last-minute edits or silent overwrites.
3. **Tamper protection** – SHA-256 is collision-resistant, making it very hard for a malicious or accidental change to go undetected.
4. **Audit trail** – Stored alongside load_id in the load ledger, allowing us to prove months later exactly which file was ingested.
5. **Fail-fast safety** – If a mismatch is found, the job aborts before staging or prod is touched.

Why we need crosswalk tables?

A crosswalk is a small lookup table that maps a **partner's source key** (e.g., source_customer_id) to **our internal primary key** (e.g., customer_id). We keep one per entity (customers, vehicles, invoices).

Why we need it

1. **Idempotency**: Rerunning the same load won't create duplicates—lookups reuse the same internal IDs.
2. **Relationship integrity**: Children (vehicles invoices line items) can reliably resolve their parents in our DB.
3. **Multipartner safety**: Different partners can reuse the same source keys; source_system disambiguates.
4. **Auditability & lineage**: We can prove when a mapping was first/last seen and which internal row it points to.
5. **Key changes/merges**: If a partner changes keys or we merge two source records, we can point multiple source keys to the same internal ID without touching prod FKs.