CanonFodder – Requirements & Design Documentation

Version 0.1 – 2025-05-19

---

1 Requirements Specification (RS)

1.1 Purpose

CanonFodder (CF) is a reproducible data-engineering pipeline that ingests music listening events (*scrobbles*), enriches them with metadata, stores them in a relational data warehouse, and offers interactive analytics and visualisation. The RS captures **what** the system must achieve, independent of the implementation.

1.2 Scope

- In scope: automated data ingestion from Last.fm, MusicBrainz enrichment, manual/automatic canonisation, storage in MySQL/SQLite/PostgreSQL, parquet export, BI dashboard, workflow orchestration with Airflow, Docker packaging.
- Out of scope: streaming playback, real-time recommendation engine, paid cloud hosting.

1.3 Definitions & Acronyms

Term	Definition
Scrobble	One track play event (artist, track, timestamp)
UTS	Unix Time Stamp provided by last.fm
CF	CanonFodder project
DWH	Data-warehouse schema in RDBMS
Eval DB	Parquet star-schema export for lightweight analytics.

AVC	artist_variants_canonized table
ETL	Extract-Transform-Load
BI	Business-intelligence frontend

1.4 Stakeholder Identification & Prioritisation

Priority	Stakeholder	Motivation
P0	Core developers / maintainers	Build & operate CF
P0	Power users / Quantified-Self music enthusiasts	Analyse personal listening history
P1	Casual last.fm users	Occasional insight into listening habits
P1	Researchers (musicology, data science)	Reliable structured dataset
P2	Integration partners (ListenBrainz, streaming platforms)	Optional data exchange
P3	Business sponsors	Potential monetisation & brand presence

1.5 Overall Description

CF is a **pull-based pipeline** triggered manually or via Airflow. It converts raw scrobbles into a clean star schema, normalising artist aliases and enriching country metadata. Users launch canonicalisation and visual exploration from a CLI menu.

1.6 Assumptions, Dependencies, Constraints

- **Python ≥ 3.12**, PEP-8 compliance, 80 % test coverage
- Default RDBMS = MySQL 8; SQLite/Postgres supported
- External APIs – last.fm (*API key*), MusicBrainz (rate-limited), public internet
- Local filesystem read/write for parquet cache
- Docker container run via docker run canonfodder

---

2 Software Requirements Document (SRD)

2.1 System Interfaces

ID	Interface	Direction	Protocol / Lib	Description
I-01	last.fm API	Inbound	HTTPS REST	user.getRecentTracks, user.getInfo
I-02	MusicBrainz API	Inbound	HTTPS REST	Artist lookup & search
I-03	RDBMS	Bidirectional	SQLAlchemy 2.0	MySQL / PostgreSQL / SQLite
I-04	File System	Outbound	Parquet (pyarrow)	Eval DB export
I-05	CLI	Inbound	argparse / questionary	User launches workflows
I-06	Airflow DAG	Inbound	Python API	Scheduled orchestration

2.2 Functional Requirements

ID	Description
FR-01	**Fetch Scrobbles** – Pull recent tracks for a user since the last stored timestamp and persist raw JSON.
FR-02	**Normalise Scrobbles** – Rename columns, convert UTS→UTC datetime, and remove duplicates.
FR-03	**Bulk Insert** – Load normalised rows into scrobble table with dialect-aware conflict ignore.
FR-04	**Artist Enrichment** – For new MBIDs, fetch country & aliases; cache in artistcountry.
FR-05	**Canonisation** – Group artist name variants, store mapping in AVC, apply to scrobble history.
FR-06	**Parquet Export** – Dump star schema to parquet files on demand or based on schedule.
FR-07	**BI Frontend** – Provide menu to launch data gathering and open interactive dashboards using custom colour palettes.

FR-08	**Retry & Back-off** – Network requests retry up-to *8* times with exponential back-off.
FR-09	**Workflow Orchestration** – One-click Airflow DAG (cf_ingest) covering FR-01–FR-07.
FR-10	**Docker Distribution** – Build image with code, dependencies, docs, dashboards.

2.3 Performance Requirements

- Capable of ingesting **10 000 scrobbles per minute** on consumer-grade hardware.
- End-to-end Airflow DAG completes within **15 min** for 1 million scrobbles.

2.4 Data Requirements

- All timestamps stored in UTC with timezone awareness (TIMESTAMP WITH TIME ZONE)
- Primary keys, unique constraints and foreign keys per ERD (see §3.4)
- Eval DB parquet partitioned by year for scrobble table

2.5 Reliability & Availability

- **Retry logic** on HTTP 5xx and network errors (see FR-08)
- Alembic migrations version-control schema
- Unit + integration tests; CI fails if coverage $< 80\,\%$ or linting errors

2.6 Security & Privacy

- Secrets stored in .env; mounted via Docker secrets in production
- Only read-only last.fm API scope
- GDPR compliance: user can *purge* all data via CLI command

2.7 Quality Attributes

- **Maintainability** – Docstrings, PEP-8, and consistent type hints
- **Portability** – Runs on Linux/Mac/Windows via Docker

- **Usability** – Guided CLI prompts; colour-blind-safe palettes

2.8 Constraints & Limitations

- Rate limits: last.fm ≈ 5 req/sec; MusicBrainz 1 req/sec. The pipeline must honour these to avoid bans.
- Offline mode requires cached parquet; no on-device scrobbling if internet absent.

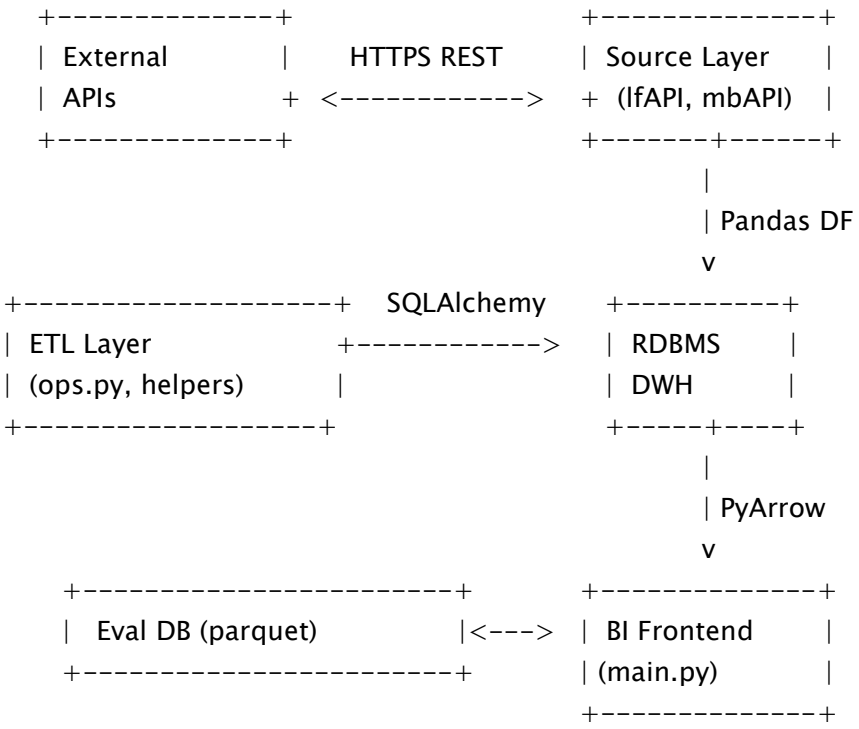
2.9 Acceptance Criteria

- All functional requirements demonstrably fulfilled.
- Ingest 100 k scrobbles without duplication.
- Dashboard renders top-10 artists within 2 seconds.

---

3 High-Level Design (HLD)

3.1 Architectural Overview



3.2 Key Components

Component	Responsibilities
Source Layer	Fetch scrobbles (lfAPI), fetch artist data (mbAPI), resilient client (HTTP/client.py)
ETL Layer	Data normalisation, canonicalisation, enrichment, bulk inserts, parquet export
Persistence	RDBMS schema; Alembic migrations; evaluation parquet store
Orchestration	Airflow DAG & CLI glue commands
BI Frontend	CLI menu & notebooks/dashboards using palettes.json

3.3 Data Flow

1. Airflow triggers **fetch & ingest** task.
2. lfAPI pulls new scrobbles, returns JSON.
3. ops.py converts to DataFrame → dedupe → bulk insert.
4. mbAPI enriches missing artist rows; bulk upsert.
5. Canonisation (helpers/cli) groups variants, updates AVC.
6. Parquet export updates Eval DB.
7. User opens dashboard to explore.

3.4 Database ER Schema (simplified)

Table	PK	Important Columns	FKs / Constraints
scrobble	id	artist_name, track_title, play_time (UNIQUE together)	artist_mbid → artistcountry.mbid (nullable)
artist_info	id	artist_name, mbid (UNIQUE), country	

artist_variants_canonized	artist_variants_hash	canonical_name, timestamp	
user_country	id	country_code, start_date, end_date (+ period check)	
ascii_chars	ascii_code	ascii_char (lookup)	
country_code	iso2	iso3, en_name, hu_name	

3.5 Technology Stack & Rationales

- **Python 3.12** – modern typing, datetime.UTC
- **SQLAlchemy 2** – declarative ORM + dialect-aware inserts
- **Airflow 2** – battle-tested workflow scheduler
- **Docker** – reproducible deployment
- **Parquet** – efficient columnar analytics

3.6 Deployment View

- Single container includes CF code + mysql:8 service via docker-compose for local use
- Optionally deploy to cloud VM; mount volume for parquet cache

---

4 Low-Level Design (LLD)

4.1 Module Breakdown

Module	Key Classes / Functions	Algorithmic Notes
lfAPI.py	fetch_recent_tracks_all_pages(), bulk_insert_scrobbles()	Pagination + signature auth; respects rate

		limit; yields DataFrames.
ops.py	_prepare_scrobble_rows(), _bulk_insert()	Vectorised Pandas ops; dialect-aware insert_ignore.
mbAPI.py	lookup_mb_for(), fetch_country()	Caches responses; Tenacity retry; adheres to 1 req/sec.
helpers.cli	unify_artist_names_cli()	Terminal prompts for manual grouping; hashes variants.
helpers.io	dump_latest_table_to_parquet()	Uses pyarrow for zero-copy parquet write.
corefunc.dataprofiler	quick_viz()	Generates seaborn-free matplotlib charts; palette from JSON.

4.2 Class Interfaces (excerpt)

```
class Scrobble(Base):
    id: int  # PK
    artist_name: str(350)
    album_title: str(350)
    track_title: str(350)
    artist_mbid: str(36)|None
    play_time: datetime(timezone=True)
```

4.3 Algorithms & Data Structures

- **Canonicalisation** – RapidFuzz string similarity + DBSCAN + manual anchors.
- **ASCII Frequency** – Pre-seed 33–126 ASCII chars; generate ascii_freq() Series for profiling.
- **Bulk Insert** – Chunks of 10 000 rows; uses *INSERT IGNORE* / *ON CONFLICT DO NOTHING*.

4.4 Error Handling & Logging

- client.make_request() retries $\times 8$; logs to stdout; non-fatal errors escalate via raise_for_status in DAG context
- Alembic migration failures abort DAG

4.5 Configuration Management

- .env parsed by dotenv; required keys validated at start-up
- Default values fallback to SQLite for tests

4.6 Testing Strategy

- **pytest** test suite in tests/; fixtures for temporary SQLite
- CI matrix: unit, integration with Dockerised MySQL, coverage gate $\geq 80\,\%$

4.7 Performance Optimisations

- Use Pandas vectorisation and conditional SQL INSERT IGNORE
- Parquet partitioning by year/month for pruning

4.8 Open Issues / Future Work

- OAuth flow for ListenBrainz
- Real-time streaming ingest
- Web-based dashboard (Streamlit) with palette auto-theming

---

5 Traceability Matrix (RS → FR)

RS Section	FR IDs
1.2 Scope – ingest, enrich, store	FR-01 … FR-04
1.2 Scope – parquet export	FR-06
1.2 Scope – BI frontend	FR-07
1.6 Constraints – Docker	FR-10