TLPTACO v2 – User Guide

> **tlptaco** (Teradata Logistics Pipeline for TArgeting and COmmunications) is a modular Python / Jinja-SQL toolkit that turns YAML configuration files into a complete *Eligibility → Waterfall → Output* campaign pipeline.

This document is the **authoritative, end-to-end "how-to"** for analysts and data engineers who need to configure, run, and extend tlptaco.

Table of Contents

Installation

tlptaco is distributed as source; clone the repository and install the Python dependencies into a virtual-environment:

```
git clone <repo>
cd tlptaco
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt   # includes teradatasql, jinja2, rich, pydantic
```

The project assumes **Python 3.10+** and connectivity to a Teradata database.

Quick Start

```
# 1 . Write a YAML config (see full reference below)
cp example_yaml.yaml my_campaign.yaml   # fill in real values

# 2 . Run the full pipeline (Eligibility → Waterfall → Output)
python -m tlptaco.cli -c my_campaign.yaml -o ./run_artifacts --progress

# 3 . Inspect logs / Excel waterfall / output files in ./run_artifacts

# 4 . Analyse logs afterwards (latest run only)
python -m tlptaco.logs -f run_artifacts/logs/tlptaco.log --summary
```

Command-Line Interfaces

1. Main pipeline runner (`tlptaco.cli`)

```
python -m tlptaco.cli -c CONFIG [-o DIR] [--mode full|presizing] [-p] [-v]

  -c / --config       YAML or JSON config file (required)
  -o / --output-dir   Root dir for reports & logs (default: cwd)
  -m / --mode         full = Eligibility→Waterfall→Output
                       presizing = Eligibility→Waterfall only
  -p / --progress     Rich progress bars
  -v / --verbose      DEBUG console output via rich
```

2. Log-inspection utility (`tlptaco.logs`)

```
python -m tlptaco.logs -f LOGFILE [-l LEVEL …] [--summary|--print]
```

```
  -l / --levels   DEBUG INFO WARNING ERROR CRITICAL (default = all)
  --summary       Show counts per log level (default when no flag specified)
  --print         Emit matching lines themselves

The parser automatically isolates the **most-recent run** using the header
inserted by tlptaco at the start of each run.
```

Configuration YAML

All configuration is validated by Pydantic models in `tlptaco.config.schema` – any schema violation aborts execution with a clear message and line/column hint.

Below is a **field-by-field breakdown**. For an "all-in" template jump to Appendix A.

Top-level keys

Key	Type	Required	Description
`offer_code`	str	No	Short code shown in progress bars & Excel filenames (default `Running`).
`campaign_planner`	str	No	Displayed in Excel header.
`lead`	str	No	Displayed in Excel header.
`logging`	Logging	Yes	
`database`	Database	Yes	
`eligibility`	Eligibility	Yes	
`waterfall`	Waterfall	Yes	
`output`	Output	Yes	
`pre_sql`	list	No	List of SQL files to run beforehand.

Logging

```yaml
logging:
  level: DEBUG | INFO | WARNING | ERROR | CRITICAL   # console & file
  file:        logs/tlptaco.log        # optional main log
  debug_file:  logs/tlptaco.debug.log  # optional DEBUG-only log
  sql_file:    logs/tlptaco.sql.log    # rendered SQL only
  sql_exclude_sections: [waterfall, output]  # sections to skip in sql_file
```

> Emoji-rich console output is enabled with `-v / --verbose` .

Database

```yaml
database:
  host: rchtera           # default
  user: my_user
  password: ${TD_PASS}    # env var or plaintext
  logmech: KRB5 | LDAP | TD2 | NONE
```

The DB credentials are consumed by `tlptaco.db.connection.DBConnection` which delegates to the official **teradatasql** driver.

Eligibility

The engine produces a *smart* table where each **check** becomes a flag column (1 = record passed the check).

```yaml
eligibility:
  eligibility_table: user_wpb.offer123_elig   # schema.table or identifier

  # 1. Conditions ************************************************
  conditions:
    main:
      BA:          # Baseline checks every record must pass
        - sql: "col1 = 1"            # check 1
          description: "must be active"
        - sql: "date_col >= CURRENT_DATE - 30"

    # Per-channel extra checks & segments
    channels:
      email:
        BA:
          - sql: "email_opt_in = 1"
        # any key besides BA is treated as *segment*
        loyalty:
          - sql: "loyalty_status = 'GOLD'"


  # 2. Tables ************************************************
  tables:
    - name: dm.cust_base
      alias: c
      join_type: ''               # omitted = first / base table
      where_conditions: "c.active = 1"

    - name: dm.cust_attrs
      alias: a
```

```
    join_type: LEFT
    join_conditions: "c.id = a.id"

# 3. Unique identifiers persisted in final table
unique_identifiers: [c.id, c.household_id]
```

Key take-aways

- **Name auto-generation** – if you omit the `name:` key under a check it is auto-filled (e.g. `email_loyalty_1`).
- **Segments** – any sub-mapping under a channel besides `BA` becomes a *segment* that later feeds the waterfall report.

Waterfall

```
waterfall:
  output_directory: reports/poc/waterfall

  # Records are counted by *groups* – each item can be a single column or
  # a list (composite key).  Order matters (rows in Excel).
  count_columns:
    - customer_id
    - [customer_id, account_id]

  history:      # optional historical comparison
    track: true
    recent_window_days: 30        # OR compare_offset_days: 7
    db_path: reports/waterfall_history.sqlite
```

The engine writes one Excel workbook per *run* plus a consolidated workbook (`offerCode_YYYY_MM_DD_HH:MM:SS.xlsx`).

Output

```
output:
  channels:
    email:
      columns:
        - c.customer_id
        - c.email
        - chk_main1
      column_types:          # optional per-column CASTs
        c.customer_id: "VARCHAR(9)"

      file_location: reports/email
      file_base_name: email_list
```

```
    output_options:
      format: csv | excel | parquet | table
      additional_arguments: {sep: "|"}   # passed to pandas writer
      custom_function: package.module:transform_df

    unique_on: [customer_id]     # dedup via QUALIFY ROW_NUMBER()

  sms: {...}

# Failed-records optional dump
failed_records:
  enabled: true
  first_reason_only: true
  file_location: reports/failed
  file_base_name: failed_rows
  output_options:
    format: parquet
```

column_types – when provided the output template renders

```
CAST(c.customer_id AS VARCHAR(9)) AS customer_id,
```

otherwise it keeps the column untouched.

Failed-records dump

When enabled the engine writes a flat file containing every record that did **not** pass all checks, annotated with the first failure reason.

Pre-SQL

```
pre_sql:
  - path: sql/prepare_tables.sql
    analytics:
      table: sandbox.tmp_prepare
      unique_counts:
        - customer_id
        - [company_id, site_id]
```

- Each file is split on semicolons (`;`).
- Optional *analytics* run simple `COUNT(DISTINCT …)` queries and log the results.

Engines in depth

1. PreSQLEngine

• Reads each `.sql` file, splits on semicolons, executes sequentially via `DBRunner.run()`. • Optional analytics perform quick `COUNT(DISTINCT …)` and log the counts.

2. EligibilityEngine

• Builds a work-table by joining the declared *tables* and computing flag columns for every BA / segment check.
• Post-run it logs total rows and `COUNT(DISTINCT …)` for each unique identifier.

3. WaterfallEngine

• For each `count_columns` group it generates two kinds of SQL (full and segment detail) and pivots the results. • Excel writer lives in `tlptaco.engines.waterfall_excel`. • Optional history stored in SQLite enables comparison tabs.

4. OutputEngine

• Renders **output.sql.j2** per channel, applies optional `column_types` casting and `CASE` template logic. • Destination:

- *format* = *table* → creates a multiset table in Teradata.
- *file formats* → fetches DataFrame then writes CSV / Parquet / Excel via `tlptaco.iostream.writer.write_dataframe()` (auto-chmod 770 and `.end` marker file).

SQL Templates

All templates are Jinja2 files under `tlptaco/sql/templates`. You can list them programmatically:

```
from tlptaco.sql.generator import SQLGenerator
gen = SQLGenerator('tlptaco/sql/templates')
print(gen.list_templates())
```

Fork/extend templates as needed – keep custom ones in a separate directory then instantiate `SQLGenerator(custom_dir)` and monkey-patch the engine in your bootstrap code.

Logging & Diagnostics

• Main & debug logs are standard text; each run starts with a header:

```
================================================================================
TLPTACO RUN START 2025-08-07 12:00:00
================================================================================
```

• Rendered SQL is saved to a *dedicated* file when `logging.sql_file` is set. Use `sql_exclude_sections` to keep the file lean (e.g. exclude “output”). • Inspect logs after the fact with the built-in parser:

```
python -m tlptaco.logs -f logs/tlptaco.log --print -l ERROR WARNING
```

Appendix A – Example full YAML

See `example_yaml.yaml` in the repository – it contains **every** configurable field with inline explanations and choice lists.

Appendix B – Permissions

When tlptaco writes any file or directory it calls `tlptaco.utils.fs.grant_group_rwx()` which:

1. Tries `os.chown(path, -1, <gid_of_cwd>)` – change *group* ownership to match the working directory.
2. Sets mode `0o770` (rwx for owner & group) so teammates can access the artefacts without manual chmod/chgrp.

If the operation fails (platform does not support it or the user lacks privileges) the error is swallowed and execution continues.

Happy targeting! 🎯