Automated Template Generation with LLM-Based Schema Inference

BACHELOR THESIS

Lea Buchner

Submitted on 2 June 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Johannes Jablonski
Prof. Dr. Dirk Rhiele

Friedrich-Alexander-Universität
Technische Fakultät

Declaration Of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 2 June 2025

License

Erlangen, 2 June 2025

Abstract

- Objective: Develop an automated system that generates JV pipeline templates from CSV files, facilitating scalable data processing without intermediate file storage.
- Methodology: Parse CSV files to extract column headers, informing the structure of the JV templates.
  - Utilize an intermediate JSON representation as a semantically rich, introspectable abstraction layer as well as for debugging purposes before converting to the final JV format.
  - Template generation was guided by the JV user documentation and informed by an internal reference (cars.jv), which served as a working example of valid JV constructs.
- Implementation:
  - Ensure the generated JV templates are valid and ready for deployment.
  - Design the system to operate entirely in-memory, eliminating the need for temporary disk storage.
- Evaluation:
  - Validate the correctness and structural integrity of the generated JV templates across a dataset of up to 10,000 CSV files, ensuring adherence to the expected JV format and schema logic.
  - Validate the correctness of the generated templates against the expected JV format.
- LLM-Based Schema Detection (Evaluated Separately):
  - Employ a locally hosted Large Language Model (LLM) to identify header rows in anomalous or irregular CSV files.
  - Output the detected schema information in JSON format.
  - Apply prompt engineering techniques to optimize the LLM's performance, avoiding the need for model fine-tuning.
  - Evaluate various LLMs and parameter configurations to determine the most effective setup for schema detection.

- Conduct testing on a separate set of up to 10,000 CSV files to validate the LLM's effectiveness.

- Conclusion:
  - The project successfully demonstrates the feasibility of generating JV pipeline templates from CSV files, achieving the goal of a fully automated, in-memory solution based on a generalized and widely applicable schema.
  - The separate evaluation of LLM-based schema detection provides insights into the capabilities of prompt-engineered models for handling irregular data structures.

Contents

List of Figures

List of Tables

1 Introduction

2 Literature Review

3 Requirements

Template

- with a csv as input generate a template for a pipeline model working with the data in the provided csv
- read the column names to access the data correctly
- basis for this template is cars.jv, including basic blocks
- for debugging purposes, first generate json then transform to jv
- in the end no tmp files should be needed/no intermediate files should be saved to local disk
- the generated model template should be a valid jv file
- test the generation with up to 10 000 csv files

LLM Part

- a llm locally hosted should detect the row with the column names in anomal csvs
- the output of the llm should be json
- use prompt engineering as a more efficient way than finetuning for generating optimal output
- evaluate which models with which parameters are best suited for the task
- test this with up to 10 000 csv files

4 Architecture

4.1 JV Template Generation Script

4.1.1 High-Level Structure

- Modular design: functionality split into purpose-specific functions, encouraging code reuse and testability
- Entry point: a `main()` function launches a Tkinter GUI as the user-facing interface for initiating processing
- Input agnostic: accepts local CSV files, URLs, or text files with URL lists

4.1.2 Core Functional Components

4.1.2.1 User Interface (Tkinter GUI)

- Simple front-end with three triggers:
  - Select CSV file
  - Enter CSV URL
  - Choose a text file of links

4.1.2.2 Data Source Abstraction

- Supports both local paths and URLs
- Dynamically assigns extractor class:

- `LocalFileExtractor`
- `HttpExtractor`

4.1.2.3 Pipeline Schema Generator

- Constructs a schema made of functional blocks, including:
  - `Extractor`
  - `TextFileInterpreter`
  - `CSVInterpreter`
  - Optional: `CellWriter` for unnamed columns
  - `TableInterpreter` with type annotations
  - `SQLiteLoader`
- Connections between blocks are directional pipes
- Output is a structured JSON object

4.1.2.4 Output Modules

- JSON Export: `save_to_json()` writes the pipeline schema to `./JsonFiles/`
- JV Conversion: `convert_json_to_jv()` turns JSON into `.jv` format

4.1.2.5 Workflow Coordination

- Central Orchestration:
  - `process_file_or_url()` integrates all steps:
    - Reads source (local or URL)
    - Infers types
    - Generates schema
    - Saves JSON and `.jv` if configured

- Batch Processing: `process_links_file()` loads URLs from a file and calls `process_file_or_url()` for each
- Utility Tools:
  - `extract_file_name()`, `to_camel_case()`, `sanitize_name()` for normalization and safety
  - `column_index_to_label()` maps index → Excel-style label

4.1.2.6 Error Handling & Resilience

- Defensive coding ensures graceful failure on:
  - Missing files
  - URL errors
  - Unparseable CSVs
  - Malformed JSON
- Logging via Python `logging` module (errors saved to `error_log.txt`)

5 Design and Implementation

5.1 CSV Parsing & Column Type Inference

Function: `infer_csv_column_types()`

- Uses `pandas.read_csv()` for loading and previewing data
- Checks for `"Unnamed"` columns, infers types (e.g., numeric, string) based on content
- Maps pandas dtypes to internal schema types via `map_inferred_type()`
- Rename lists `renamed_cols`, `new_names` maintain consistency

Refactor no global variables

5.2 Schema Construction

Function: `generate_pipeline_schema()`

- Creates a list of blocks that define ETL logic
- Dynamically includes or skips certain blocks based on input characteristics
- Handles structural naming (e.g., column names, file names) using utilities like `sanitize_name()`
- All schema relationships defined explicitly via directional `pipes`

5.3 JSON and JV Output

JSON Export

- `save_to_json()` serializes pipeline to a JSON structure
- Ensures file creation with `os.makedirs(..., exist_ok=True)`
- Filename formatting via `to_camel_case()` ensures readability and consistency

JV Conversion

- `convert_json_to_jv()` handles:
  - Scalar fields (e.g., `"type": "string"`)
  - Lists (e.g., `"pipes": [...]`)
  - Nested dictionaries
- Outputs JV syntax in `.jv` files, saved under `./JvFiles/`

5.4 GUI Integration

Function: `main()`

- Initializes and launches a basic Tkinter window
- Provides three buttons:
  - Select local file
  - Enter URL
  - Load list of URLs
- Each button triggers its respective handler to invoke the core pipeline functions

5.5 Defensive Design Principles

- Try/Except blocks guard file operations, schema generation, and format conversion

- Centralized logging ensures all errors are visible and traceable
- Default behaviors (e.g., setting unknown columns to `"text"`) prevent crashes on ambiguity
- File safety: checks for null paths, invalid types, and filename conflicts

5.6 Design Priorities

- Extensibility: New interpreter or loader blocks can be added with minimal change
- Maintainability: Functions have single responsibilities, simplifying testing and updates
- User-Friendly: GUI abstracts complexity for non-technical users
- Robustness: Defensive code ensures smooth handling of malformed or unexpected input

6 Evaluation

7 Conclusions

Bibliography

Bill Of Materials

External Dependencies

- pandas: CSV parsing, type inference
- tkinter: GUI frontend
- json, os, pathlib: file I/O and serialization
- urllib.parse: URL type detection and validation