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

2.1 Prompt Engineering

- Lower Cost and Resource Requirements: Running a prompt is less resource-intensive than fine-tuning a Large Language Model (**LLM**) [PT24, Shi+25]. Fine-tuning requires significant resources and computing power, which can be expensive [PT24]. Prompt engineering is generally more accessible and practical, especially in environments with limited resources [PT24].
- No Requirement for Labeled Datasets: Prompting has advantages over fine-tuning because it does not require labeled datasets, which are costly to acquire [PT24, Shi+25]. Prompt engineering can adapt pre-trained language models without needing a supervised dataset [PT24].
- Accelerated Application Development: LLMs leveraged through prompt engineering can speed up application development, achieving decent performance [Shi+25, TC24]. It facilitates the building of AI systems without the need for ML model training or supervision [Shi+25].
- Avoids Fine-Tuning Issues: Prompt engineering reduces the need for expensive computational costs and challenges like catastrophic forgetting associated with fine-tuning specialized models [Shi+25]. Catastrophic forgetting occurs when models lose some of their previously learned skills while learning new domain-specific information [Shi+25].
- Decent or Even Superior Performance in Certain Scenarios: While fine-tuning generally leads to better performance [PT24], the performance achieved with prompt engineering is remarkable considering no specific training is conducted [PT24]. In some cases, such as medical Question & Answer tasks with Open-Source models, prompt engineering alone can outperform fine-tuning [Zha]. GPT-3.5-turbo significantly outperformed other approaches in specific multi-party dialogue tasks when using a 'reasoning'

style prompt in a few-shot setting [Add+23]. Prompt Engineering, when effectively harnessed, can even outperform fine-tuned specialized models like PubMedBERT for tasks such as identifying metastatic cancer [Zha]. It can also help leverage the versatile capabilities of LLMs [PT24].

- Robustness and Flexibility: Prompt engineering offers flexibility [PT24]. GPT-4 demonstrated remarkable resilience when using prompt engineering for metastatic cancer identification, maintaining performance even when keywords or a significant percentage of tokens were removed from the input text [Zha]. Using concise output instructions can facilitate downstream processing [Zha]. Different prompting techniques (zero-shot, few-shot/in-context, role-playing, chain-of-thought, task-specific, conversational) can be applied in zero-shot or few-shot settings to guide the model [Add+23, Mah24, Shi+25, TC24, Zha]. Conversational prompting, which includes human feedback, shows potential for improving results [Shi+25].

Requirements
IEEE 830-style requirements engineering
Architecture
IEEE 42010 (architecture description)
C4 model (Context → Container → Component → Code)
RUP and TOGAF methodologies

3 Requirements

3.1 JV Template Generation Script

3.1.1 Functional Requirements

FR-T1: Generate a pipeline model template from a CSV input. FR-T2: Read column names from the CSV to access the data correctly. FR-T3: Use cars.jv as a base, including predefined basic blocks. FR-T4: For debugging, first generate an intermediate .json, then convert it to .jv. FR-T5: Ensure the generated model template is a valid .jv file.

3.1.2 Non-Functional Requirements

NFR-T1: The process should not save any intermediate or temporary files to local disk. NFR-T2: Test template generation with up to 10,000 CSV files.

3.1.3 Constraints & Assumptions

C-T1: Template structure must align with JV user documentation.

3.2 LLM based Header Detection

3.2.1 Functional Requirements

FR-L1: A locally hosted LLM should detect the row containing column names in malformed or anomalous CSVs. FR-L2: Output from the LLM should be in .json format. FR-L3: Apply prompt engineering techniques to generate optimized output. FR-L4: Evaluate various models and parameter combinations to determine best fit for the task.

3.2.2 Non-Functional Requirements

NFR-L1: The LLM component should support evaluation across up to 10,000 CSV files. NFR-L2: Prefer prompt engineering over fine-tuning for performance and efficiency.

3.2.3 Constraints & Assumptions

C-L1: The LLM must be hosted locally (no external API or cloud dependency). C-L2: Output format is strictly JSON (no .jv conversion at this stage).

3.2.4 Old

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 Purpose and Problem Context

- Automate schema generation for heterogeneous CSV data sources
- Unify local and remote CSV ingestion workflows
- Enable visual pipeline representation (.jv format) from CSV structure
- Address lack of metadata structure in arbitrary CSV datasets
- Facilitate JSON-to-JV transformation for domain-specific pipeline engines

4.1.2 Architectural Style

- Component-based modular design
- File-driven pipeline orchestration
- Implicit dataflow-style execution (simulates ETL)
- Monolithic script with service emulation via functions
- GUI + batch CLI hybrid interaction (via tkinter + os.environ)

4.1.3 Core Components

- GUI: Provides a simple UI for input selection and log display
- Input Management: Loads one or more CSV files (local or from URLs)
- Schema Inference Engine: Analyzes CSV data to infer column data types

- Pipeline Generator: Transforms inferred schema into a structured pipeline schema (JSON)
- JSON & JV Writer: Converts pipeline into exportable JSON and JV files
- Naming/Path Utils: Ensures all names and paths are valid for use in code & filenames

4.1.4 Execution Environment

- Platform: Cross-platform but tested best on local environments (Windows/Mac/ Linux)
- Runtime: Python 3.x, with pandas, tkinter, logging, json, urllib, pathlib
- I/O: GUI-based file selection, file I/O, internet access for downloading URLs
- Users: Likely data engineers or technical users needing to transform CSVs into domain-specific pipeline configurations

4.1.5 Interaction Diagrams

- System-Level Data Flow Diagram (DFD): CSV input → Inference → Pipeline blocks → JSON → JV
- Service Interaction Flow (Sequence Diagram): User input → File parsing → Block generation → Pipe connection → Output writing
- Component Diagram: Logical grouping: Parsing | Transformation | Output | GUI

4.1.6 Quality Attributes

- Modifiability: Function-level isolation; extensible block system
- Scalability: Handles batch processing via link file ingestion
- Fault Tolerance: Logging mechanism for error tracking
- Testability: Deterministic, file-based inputs and outputs

- Performance: Efficient pandas-based type inference
- Security: Input sanitization for filenames and URLs

4.1.7 Constraints & Trade-offs

- Platform: Desktop-only (Tkinter), no web or headless support
- Latency: Limited optimization; not suitable for real-time processing
- Security: No sandboxing or validation for remote URLs
- Flexibility vs. Simplicity: Static block definitions hardcoded
- No plugin system: Cannot dynamically extend pipeline block types

4.2 Header Row Inference Script (LLM-based)

4.2.1 Purpose and Problem Context

- Automatically identify header row in noisy or metadata-heavy CSV files
- Leverage LLMs to perform pattern recognition and contextual reasoning on CSV preambles
- Generate structured JSON responses for downstream pipeline alignment
- Address ambiguity in human-generated CSVs with irregular preambles
- Replace brittle heuristics with a few-shot learning approach via prompt engineering

4.2.2 Architectural Style

- LLM-backed inference-as-a-service

- Stateless function-driven processing pipeline
- Prompt engineering for structured output generation
- Few-shot learning to enhance LLM prediction quality
- Local-first OpenAI-compatible API gateway integration

4.2.3 Core Components

- CSV Loader: Loads raw text content of CSVs for analysis
- Prompt Builder: Constructs few-shot prompt by concatenating examples and input
- OpenAI Client Interface: Interfaces with locally hosted OpenAI-compatible model server
- Response Validator: Extracts JSON and validates against a pydantic schema
- Error Handling Module: Catches and logs validation or communication errors

4.2.4 Execution Environment

- Platform: Any OS with Python 3.x and internet/local model access
- Runtime: Python 3.10+ recommended
- Libraries: openai, pydantic, re, langchain_core, urllib, pathlib, threading
- Users: Developers or data engineers validating CSV structure before ingestion

4.2.5 Interaction Diagrams

- System-Level Data Flow Diagram:

CSV file → Read & Preview → Prompt Construction → LLM Inference → JSON Validation → Output Service Interaction Flow (Sequence Diagram): User selects file → File read → Prompt built → API call → LLM returns JSON → Schema validation Component Diagram: Core: Loader | Prompting | LLM Interface | Output Parser

4.2.6 Quality Attributes

Accuracy: Boosted via few-shot examples Explainability: LLM explains rationale per output Fault Tolerance: Graceful fallback and raw logging Modifiability: Easily extendable examples or model parameters Interoperability: JSON response integrates with downstream pipelines

4.2.7 Constraints & Trade-offs

Model Dependency: Requires powerful language model (e.g., GPT-4) to perform well Latency: Dependent on LLM processing time and local server speed Scalability: Designed for single-file analysis, not massive batch jobs Security: Raw file content passed to external/ local model endpoint Transparency: Determinism not guaranteed; different completions possible per run

5 Design and Implementation

5.1 JV Template Generation Script

5.1.1 High-Level Design (HLD)

modular ETL (Extract-Transform-Load) pipeline generator and converter from CSV/ URL inputs to .json and .jv representations.

5.1.2 Components Responsibilities

- GUI Component:
  - Accept user inputs via file dialog
  - Trigger processing sequence
  - Display progress logs
  - Lock interface during execution
- Input Management Component:
  - Load CSV files from local or remote sources
  - Route to appropriate loader
  - Normalize data for downstream processing
- Schema Inference Component:
  - Analyze CSV structure
  - Infer data types for each column
  - Handle unnamed columns
- Pipeline Generation Component:
  - Map inferred schema to pipeline blocks

  - Assign IDs and structure logical flow
  - Generate metadata and config for downstream use
- JSON & JV Writer Component:
  - Serialize pipeline schema to JSON
  - Convert JSON to custom JV format
  - Ensure file naming and path validity
- Naming/Path Utilities Component:
  - Clean and format strings for use in code
  - Generate valid identifiers
  - Ensure OS-safe filenames and paths

5.1.3 Data Flow

- Event-driven invocation (via GUI)
- Linear stage transitions: Input → Schema → Pipeline → Output
- Intermediate representation: memory-held schema & pipeline
- File-based persistence

5.1.4 I/O Formats & Schemas

- Input: CSV from file or HTTP
- Intermediate format:
  - Normalized schema: `[ { name, type }, … ]`
  - Pipeline JSON: blocks, pipes, metadata
- Output:
  - `.json`: serialized config
  - `.jv`: custom line-based schema

5.1.5 Technology Stack

(as architectural decision, not implementation detail)

- Chosen stack:
  - Language: Python 3.x
  - GUI: Tkinter (desktop-focused)
  - Data: Pandas for inference abstraction
- Rationale:
  - Broad OS support
  - Familiar libraries for rapid development
  - Script-oriented architecture for flexibility
- Alternatives considered:
  - Java: Too heavyweight for a simple script
  - C++: Overkill for the task, complex GUI handling
  - Node.js: Not suitable for desktop GUI applications
- Future considerations:
  - Potential for web-based GUI if needed
  - Modularization for microservices architecture

5.1.6 Error Handling

(Design-Level View)

- Validation at input boundary (file/URL)
- Fallback types for ambiguous columns
- Logging architecture for tracking issues
- GUI interlocks for bad states
- Separation of concerns prevents error cascade

5.2 Implementation Details

5.2.1 Component Implementations

- GUI (`tkinter`)
  - `Tk()` window with `askopenfilename` and `ScrolledText` widget
  - Callback binds: `Run` button triggers main logic
  - UI lock/unlock using `state=DISABLED/ENABLED`
  - Uses `threading.Thread` to prevent UI freezing
- Input Handling
  - Local file: `open(file, 'r')` with encoding fallback
  - Remote file: `urllib.request.urlopen()` with `BytesIO` fallback
  - Batch mode: detects `.link` file, iterates over entries
- Schema Inference
  - `pandas.read_csv()` with `nrows=1000`
  - Column dtype mapping logic:
    - `object` → Text
    - `float64` → Number
    - Simple type reduction via `df[col].apply(type).nunique()`
    - Empty/ambiguous columns dropped
- Pipeline Generator
  - Constructs dictionary with:
    - `blocks[]` from column names/types
    - `pipes[]` connecting input → block → output
    - `meta{}` for title, timestamp, etc.
  - Static block templates (e.g. `{'type': 'NumberBlock', 'params': {...}}`)
- Output Writer
  - `json.dump()` for `.json`

  - `.jv` format: line-by-line write with tabular key-value representation
  - `safe_name()` used for filenames
  - Output folder auto-created with `os.makedirs()` if needed
- Naming & Path Utilities
  - `safe_name(name)` sanitizes with `re.sub()` for unsafe chars
  - Uses `Path(...).resolve()` for consistent path handling
  - Converts spaces, special characters in column names

5.2.2 Control & Data Flow

- Single-threaded logic outside GUI callbacks
- Sequential:
  1. Load CSV
  2. Infer schema
  3. Build pipeline
  4. Write outputs
- Logging via `logging.info()` and `ScrolledText` redirect

5.2.3 Interfaces in Code

- Functions:
  - `load_csv(path_or_url)`
  - `infer_schema(df)`
  - `generate_pipeline(schema)`
  - `write_json(config)`
  - `write_jv(config)`
- Data contracts: Python dicts passed between steps
- Temporary in-memory formats only, no DB/cache

5.2.4 Technology Stack (Code Usage)

- `tkinter` for GUI
- `pandas` for CSV parsing & type inference
- `urllib`, `io.BytesIO` for remote input
- `logging`, `threading`, `pathlib`, `json`, `re`, `os`

5.2.5 Error Handling

- `try/except` around all I/O points
- Logs error with traceback to GUI log pane
- Missing files: show GUI popup or log warning
- Type inference fallback to "Text" if uncertain
- GUI shows status: success, failure, or warnings

5.2.6 Misc Implementation Notes

- No unit tests, but deterministic logic for testability
- Static templates mean no dynamic codegen or plugin loading
- Script architecture (vs package) for ease of use
- CLI batch mode inferred from `.link` file extension

5.3 Header Row Inference Script

5.3.1 High-Level Design (HLD)

A CSV schema-header row inference tool powered by a local OpenAI-compatible LLM and structured prompt engineering. Outputs are validated using pydantic models.

5.3.2 Components Responsibilities

- CSV Loader:
  - Opens local CSVs and reads the first 20 lines
  - Supports files with various encodings
- Prompt Builder:
  - Uses examples to construct a few-shot prompt
  - Formats request strictly per schema requirements
- LLM Client (OpenAI):
  - Connects to a local OpenAI-compatible API endpoint
  - Sends prompts with system/user message roles
  - enforces JSON output
- Output Parser (LangChain + Pydantic):
  - Uses a JsonOutputParser with the Header model
  - Extracts JSON from potentially noisy responses
  - Validates field types (columnNameRow, Explanation)
- Error Handler:
  - Wraps API and parsing in try/except block
  - Logs or prints informative error and fallback data

5.3.3 Data Flow

Input → Preview Extraction → Prompt Generation → LLM Inference → JSON Extraction → Validation → Result Output

5.3.4 I/O Formats & Schemas

Input: Raw CSV lines (first 20) Prompt: Few-shot prompt string with embedded examples Expected Output: { “columnNameRow”: 3, “Explanation”: “The row containing column headers follows non-pattern metadata rows.” } Schema (Pydantic): Defined in header_pydantic.Header model Requires both columnNameRow (int) and Explanation (str)

5.3.5 Technology Stack

Language: Python 3.10+ LLM Backend: Locally hosted GPT-compatible endpoint (OpenAI API interface) Prompt/Output: LangChain’s JsonOutputParser Pydantic for schema enforcement Tools Used: openai (via openai.Client) re, json, pydantic, langchain_core.output_parsers Rationale: Local model for privacy and speed Pydantic for strong validation LangChain to simplify structured extraction

5.3.6 Error Handling

All I/O and API calls are wrapped in exception handling Fallbacks: Missing JSON → raises ValueError Validation error → raises ValidationError with traceback Raw LLM output preserved for debugging Uses GUI log or console print for errors and responses

5.4 Implementation Details

5.4.1 Component Implementations

CSV Loader load_csv_as_text(path) opens and reads lines using ‘utf-8-sig’ encoding Prompt Builder build_prompt(csv_20_lines) formats structured prompt with 3 few-shot examples Encodes example content, reference answers, and schema rules LLM Inference client.chat.completions.create() with: Model: “gpt-4” Role: “system” and “user” Format: {“type”: “json_object”} Temperature: 0 (deterministic) Timeout: 60s JSON Parser extract_json(text) uses regex to pull JSON block from LLM output parser.parse(json_str) validates against schema Schema Validation Uses Header pydantic class Enforces int/string types and key presence Error Catching Print tracebacks and raw output Graceful degradation for debugging

5.4.2 Interfaces in Code

Functions: load_csv_as_text(path) build_prompt(csv_str) extract_json(response_text) Model Usage: Header pydantic class LangChain JsonOutputParser(pydantic_object=Header) Data Contracts: Input: string of 20 lines Output: validated JSON object

5.4.3 Control & Data Flow

Single-run flow with no persistent state Sequence: Load file Build prompt Call API Extract JSON Validate and print

6 Evaluation

7 Conclusions

Bibliography

[PT24] C. Pornprasit and C. Tantithamthavorn, "Fine-tuning and prompt engineering for large language models-based code review automation," *Information and Software Technology*, vol. 175, p. 107523, 2024, doi: https://doi.org/10.1016/j.infsof.2024.107523.

[Shi+25] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code." [Online]. Available: https://arxiv.org/abs/2310.10508

[TC24] F. Trad and A. Chehab, "Prompt Engineering or Fine-Tuning? A Case Study on Phishing Detection with Large Language Models," *Machine Learning and Knowledge Extraction*, vol. 6, no. 1, pp. 367–384, 2024, doi: 10.3390/make6010018.

[Zha] V. S. A. S. W. J. M. H. B. M. S. L. D. D. A. J. D. W.-L. M. C. D. Z. J. C. B. Zhang X Talukdar N, "Comparison of Prompt Engineering and Fine-Tuning Strategies in Large Language Models in the Classification of Clinical Notes," *AMIA Joint Summits on Translational Science Proceedings*, pp. 478–487.

[Add+23] A. Addlesee, W. Sieińska, N. Gunson, D. H. Garcia, C. Dondrup, and O. Lemon, "Multi-party Goal Tracking with LLMs: Comparing Pre-training, Fine-tuning, and Prompt Engineering." [Online]. Available: https://arxiv.org/abs/2308.15231

[Mah24] G. A. S. N. e. a. Maharjan J., "OpenMedLM: prompt engineering can outperform fine-tuning in medical question-answering with open-source large language models.," *Scientific Reports*, vol. 14, no. 1, p. 14156, 2024, doi: https://doi.org/10.1038/s41598-024-64827-6.

Acronyms

LLM Large Language Model

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