Jayvee 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

Jayvee is a domain-specific language (**DSL**) for defining and running data pipelines. This thesis explores the generation of Jayvee templates from CSV files and the evaluation of locally hosted Large Language Models (**LLMs**) for schema detection.

The template generation is aimed at users with little programming experience, allowing them to create data pipelines without extensive coding knowledge. When working with CSV files, users often encounter challenges such as irregular header rows, metadata-heavy preambles, and inconsistent formatting. Some of those irregularities can be addressed using simple Regular Expressions (**RegEx**) or heuristics, but they may not be robust enough for all cases. Therefore, this thesis proposes a solution that leverages **LLMs** to identify the row containing the column names in noisy or actually any CSV file, followed by generating structured JavaScript Object Notation (**JSON**) responses for downstream processing. Since the template generation system uses the column names to access the data in the CSV files, the focus of evaluating the **LLMs** is on the detection of the column name row.

Evaluated were LLaMA 3 (Meta, 70B), Mistral (Mixtral MoE, 8×7B active, 46.7B total), Google Gemma 3 (12B), Google Gemma 1 (7B Instruct), Alibaba Qwen 2.5 (32B), Microsoft Phi-3 (14B), and DeepSeek R1 (≈14B distilled) as most promising, state of the art models. The evaluation was performed on a set of 1000 CSV files, which were generated from the original CSV files by adding random noise to the header rows and preambles. The evaluation led to the conclusion, that extensive processing capabilites are needed to locally host **LLMs** that are capable of this task at the current state of technology. Smaller Models (mistral:7b and Llama2 7b hf) without the necessity of a Graphics Processing Unit (**GPU**) (running on Central Processing Unit (**CPU**) only) performend badly on complex CSV files. The lack of trainingsdata (to finetune a **LLM** at least 1k and 2k of examples are needed to achieve a performance deterioration [1]) for the task of schema detection lead to the approach of using prompt engineering to optimize the performance

of the **LLMs**, which did improve the output quality significantly. The template generation system without any **LLM** inference is able to handle well formed data and even handles some anomalies correctly by scipping non parsable lines as long as the column names are parsable and thus the data can be accessed.

Contents

List of Figures

List of Tables

1 Introduction

In this thesis, the generation of Jayvee templates from CSV files with a locally hosted LLM for shema inference is explored. Focus is the header detection in irregular datasets without relying on extensive user input or pre-cleaned files. Studies indicate that prompt-engineered **LLMs** can effectively detect column headers in noisy CSV files, surpassing traditional heuristic methods in real-world applications [Lei24, Yua24]. A study on prompt engineering for LLMs demonstrates improved performance over traditional heuristic methods in processing complex tabular data, achieving accuracy gains of 0.8% to 4.07%. The traditional approaches evaluated include random sampling, evenly spaced sampling, and content snapshot sampling—all heuristic-based techniques. [Yua24]

1.1 Jayvee Template Generation

For the Jayvee Template Generation, a Python script is developed to automate the process of generating templates from CSV files. It infers the column types, renames columns if needed and generates a JSON schema that defines the extraction, interpretation, and loading of data into a SQLite database. The program supports handling local files, URLs, or lists of URLs from a text file. The schema is then converted to Jayvee. The program also includes a Graphical User Interface (**GUI**) built with Tkinter to let users select files or enter URLs easily. Errors during reading or processing are logged, and the tool supports batch processing of many CSV links. Overall, this tool streamlines the pipeline creation and data ingestion steps needed for tabular data engineering projects.

1.2 LLM Schema Inference

The schema inference covers the detection of the header row in anomalous CSV files. This referrs to the case where the first few rows of the provided file contain random metadata or comments. Since there's no standard way to identify the header row in such cases, the inference is done using a locally hosted **LLM**. Different Models were evaluated, including DeepSeek-R1 granite-3.2-8b-instruct and Qwen3-235B-A22B. The evaluation was performed on a set of 1000 CSV files, which were generated from the original CSV files by adding random noise to the header rows and preambles.

The table below summarizes the selected models used in the evaluation, followed by a detailed description of each.

Model Family	Selected Model	Reason
LLaMA (Meta)	LLaMA 3 - 70B	Top-class on general benchmarks (math, logic); noisy input resilience via long context; very large (slow) inference
Mistral	Mixtral (MoE 12.9B active)	Outperforms Llama2-70B with 6× faster inference; supports many languages; cost-efficient sparse MoE (12.9B active)
Gemma (Google)	Gemma 3 - 12B	Beats much larger models on human-eval; function-calling for JSON/schema; quantized versions for speed
Qwen 2.5 (Alibaba)	Qwen 2.5 - 32B	State-of-the-art on structured-output tasks; "understands tables"; huge context (up to 1M) enables full-file inputs
Phi-3 (Microsoft)	Phi-3 - 14B	Outperforms much larger GPT-3.5 on reasoning; very efficient (ONNX/DirectML optimized); ideal for on-device/low-latency use
DeepSeek	DeepSeek R1 - 13B distill	RL-trained reasoning model; matches GPT-3.5 on complex reasoning; focused on novel inference skills (e.g. pattern discovery) but less tested on standard tasks

Table 1: Selected Models for Evaluation

Meta's LLaMA 3 (70B) is an open large language model with strong reasoning abilities and support for long inputs, up to 128K tokens. It uses grouped query attention and a new positional encoding scheme for improved efficiencywith long sequences. Although not specifically benchmarked on CSV/header extraction, it performs well on general reasoning tasks (e.g. logic, math) and handles unstructured or messy inputs robustly. Due to its size, it requires significant compute for inference but is often ranked among the top-performing open models. [Sah24]

Mixtral (8×7B, 12.9B active) is a Mixture-of-Experts model developed by Mistral. Only 2 out of 8 expert subnetworks are active per token, so despite having 46.7B total parameters, it operates with the cost of a 12.9B model. It supports 32K-token inputs and performs well on logic and code tasks, making it suitable for structured data like tables.

It offers efficient inference and has been shown to outperform some larger models (e.g., LLaMA 2 70B) on various benchmarks. [tea23]

Google's Gemma 3 (12B) supports inputs up to 128K tokens and includes built-in functionality for generating structured outputs such as JSON. Although it is smaller than some models, it performs competitively in human evaluation rankings and is available in optimized formats for faster inference. These characteristics make it useful for processing large or complex documents, including CSV files. [Cle25, Tea25]

Alibaba's Qwen 2.5 (32B) is designed specifically with structured and tabular data in mind. Some variants, like Qwen 2.5-14B, support extremely long inputs (up to 1 million tokens). It includes functionality for structured output generation (e.g., JSON schemas) and performs well in instruction following, code, and math benchmarks. Its design makes it suitable for reading and interpreting large CSV files. [Clo25]

Microsoft's Phi-3 (14B) supports 128K-token inputs and has been optimized for efficiency, including compatibility with ONNX and DirectML for faster inference. It shows strong results in logic and math tasks, often performing better than larger models. Phi-3's ability to handle long documents with low latency makes it a good option for large-scale structured data processing. [Bil24]

DeepSeek R1 (14B) is trained through reinforcement learning to focus on logical and mathematical reasoning. While it is not explicitly tested on CSV-related tasks, its strong performance on code and logic suggests potential for handling structured data. The models are still in early-stage development and mainly focused on research, but they show capabilities comparable to larger proprietary systems on reasoning benchmarks. [Dee25]

2 Literature Review

2.1 Prompt Engineering

- Lower Cost and Resource Requirements: Running a prompt is less resource-intensive than fine-tuning a **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
Inference-as-a-Service = exposing a model's prediction/inference capabilities over a network via stateless API endpoints

We observed that few-shot with reflection & prompt decomposition achieved the best accuracy, while techniques such as few-shot and CoT achieve a 75% accuracy on our evaluation dataset. https://medium.com/@fhuthmacher/beyond-rule-based-data-quality-exploring-llm-powered-anomaly-detection-9a0c7c98c690

3 Requirements

3.1 JV Template Generation

3.1.1 Functional Requirements

FR-T1: Generate a pipeline model template from CSV input.
FR-T4: As an abstraction layer, first generate an intermediate .json, then convert it to .jv.
FR-T5: Ensure the generated model template is a valid .jv file.
FR-T7: Both local CSV-Files and remote URLs should be handled correctly.
FR-T8: Regular anomalies like leading white spaces/unnamed columns should be ignored.

3.1.2 Non-Functional Requirements

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 Schema Inference

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 model and parameter combinations to determine best fit for the task.

3.2.2 Non-Functional Requirements

NFR-L1: Evaluate correct schema inference across up to 10,000 CSV files.

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.

4 Architecture

4.1 JV Template Generation

The JV Template Generation Script was developed to automate schema generation from heterogeneous CSV sources, addressing the need to process and unify both local and remote datasets within a standardized pipeline model. This utility facilitates the transformation of arbitrary CSV data into structured, visualizable pipelines (.jv format), thereby bridging the gap between unstructured tabular data and domain-specific pipeline execution engines. A key motivation behind this tool is the lack of consistent metadata in CSV datasets, which complicates downstream ETL and pipeline configuration processes.

At its core, the architecture follows a component-based, modular design philosophy. Execution is file-driven, simulating an implicit dataflow-style execution pattern common in ETL tools. While architected as a monolithic Python script, the tool emulates service-like boundaries through modular functions. It operates in a hybrid interaction model, combining a lightweight GUI for user-friendly operations and CLI compatibility for automated or batch processing scenarios.

4.1.1 Core Components

The JV Template Generation Script is composed of several tightly integrated architectural components, each designed to handle a distinct responsibility within the pipeline generation lifecycle. At the front end, a minimal graphical user interface (GUI) built with tkinter provides users with an intuitive mechanism for selecting CSV files, whether stored locally or accessed via URLs, while also displaying processing logs for transparency. This GUI feeds into the Input Management subsystem, which is responsible for loading one

or more CSV files using standard I/O or network retrieval via urllib, thereby supporting both local workflows and remote data sourcing.

Once data is ingested, control passes to the Schema Inference Engine. a lightweight analytics layer that uses pandas and heuristic rules to detect and infer column-level data types and patterns across the CSV dataset. This inferred schema serves as the foundation for the Pipeline Generator, a transformation module that converts the columnar structure into a structured, domain-aligned pipeline schema, represented internally as a JSON object. This abstraction models the data flow logic and sets the stage for export.

Subsequently, the JSON & JV Writer modules serialize this pipeline schema into two primary output formats: a JSON version used for inspection and interoperability, and a .jv file tailored for consumption by domain-specific pipeline engines that expect this format. Underpinning all these layers is a suite of Naming and Path Utilities, which enforce naming conventions, sanitize file paths, and ensure filesystem-safe identifiers across the pipeline lifecycle. These components collectively form a streamlined, extensible toolchain that transforms raw CSV data into structured, executable pipeline blueprints.

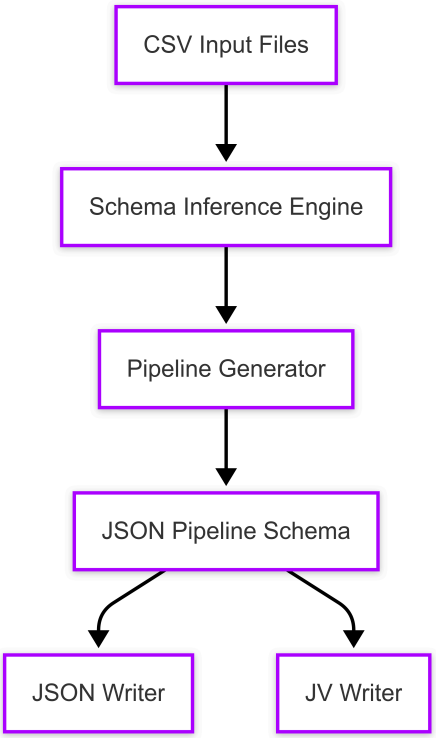

Figure 1: System-Level Data Flow Diagram – From CSV Ingestion to JV Output

These components work together to deliver a seamless transformation from raw, unstructured CSV data into structured, executable pipeline definitions. To better understand how data moves through the system, Figure 1 (see the System-Level Data Flow Diagram) provides a high-level overview of the end-to-end process, beginning with CSV ingestion and culminating in final output formats, JSON and .jv. Initially, raw CSV input files are processed by the Schema Inference Engine, which analyzes the data structure to infer column data types. This inferred schema is then passed to the Pipeline Generator, responsible for constructing a logical and structured representation of the pipeline, known as the JSON pipeline schema. Serving as the authoritative model for the pipeline, this JSON schema forms the foundation for serialization. Dedicated writer components subsequently convert this schema into two distinct outputs: a human-readable JSON file for inspection and interoperability, and a .jv file tailored for domain-specific pipeline execution engines. This data flow demonstrates the implicit ETL-like behavior of the script and highlights its

transformation-oriented architecture, where each stage builds upon the previous output to create a standardized, interoperable pipeline configuration.

4.1.2 Execution Environment

The script runs across Windows, macOS, and Linux, with optimal performance observed in local development settings. Written in Python 3.x, it depends on core libraries including pandas, tkinter, json, urllib, pathlib, and logging. It interacts with both file systems and web endpoints to load CSVs and export structured files. The typical user persona is a data engineer or technical analyst working to operationalize CSV-based data inputs into pipeline-compatible formats.

4.2 LLM Schema Inference

The architecture for LLM-schema inference can be viewed as a multi-component LLM inference system designed to serve large language models (LLMs) efficiently to clients via a standard API interface. It includes three main layers:

Layer	Role	Key Components
Client Layer	API consumers	Python scripts, LangChain agents, external apps using OpenAI API
Inference Layer	Model serving & inference	vLLM API server (OpenAI-compatible), running on GPU nodes
Orchestration Layer	Resource scheduling & job management	SLURM workload manager, cluster nodes

Table 2: Architecure Overview

The system architecture is organized into three distinct layers that interact to provide a scalable, OpenAI-compatible large language model (LLM) inference service.

The client layer is responsible for offering users a familiar and standardized interface for sending inference requests in OpenAI API formats such as /v1/completions and /v1/chat/completions. This layer abstracts the underlying model backend, ensuring a seamless "LLM-as-a-Service" experience, which allows users to interact with the models without needing to understand the complexities of model deployment or GPU resource management.

The inference layer serves as the core of the system, hosting the LLM, for example, the Qwen2.5-1.5B model, on GPU nodes and exposing REST API endpoints compatible with the OpenAI API standards. This layer supports advanced features such as batching, streaming responses, and optimized memory management, which enable efficient and performant inference even for large and complex requests.

Finally, the orchestration layer dynamically manages GPU resources by leveraging SLURM, scheduling the vLLM server process as a SLURM job within a shared GPU cluster. This approach ensures fair resource allocation across multiple users and workloads and supports horizontal scalability by launching additional SLURM jobs as demand increases.

Together, these layers form a robust and efficient system that delivers an accessible LLM inference service to users while abstracting and managing the underlying infrastructure and resource allocation transparently.

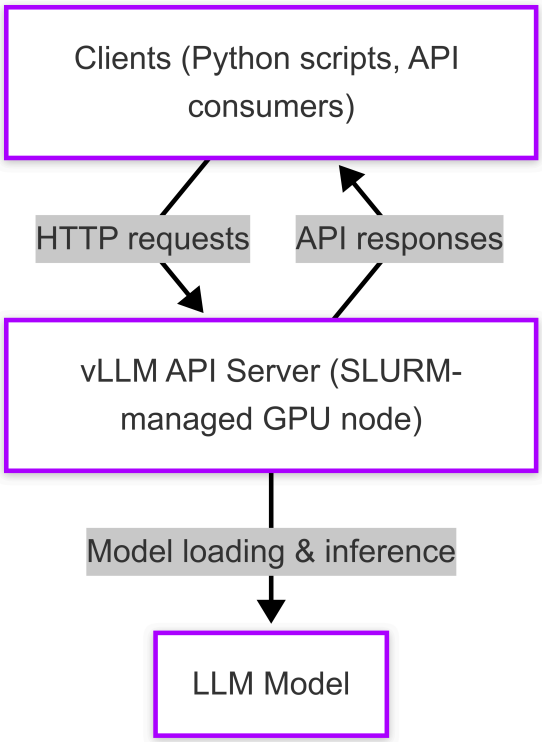

Figure 2: Data Flow Diagram

The data flow within this architecture is illustrated in Figure 2, where the Client Layer sends inference requests to the Inference Layer via OpenAI-compatible REST API endpoints, ensuring a standardized and user-friendly interface. The Inference Layer, responsible for executing these requests, manages model loading, batching, and memory optimization on GPU nodes, facilitating efficient large-scale inference. The Orchestration Layer operates behind the scenes, leveraging SLURM to dynamically allocate and manage GPU resources by scheduling the vLLM server processes as jobs on a shared GPU cluster. This setup enables fair resource distribution, supports horizontal scalability, and maintains system responsiveness even under varying workloads.

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 LLM Schema Inference

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
- Output Schema: { “type”: “object”, “properties”: { “columnNameRow”: {“type”: “integer”}, “Explanation”: {“type”: “string”} }, “required”: [“columnNameRow”, “Explanation”] }

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: "deepseek" /others to be evaluated
  - 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

6.1 JV Template Generation Evaluation

When evaluating the robustness of a CSV type inference system that incorporates both column name and content heuristics, a number of non-trivial edge cases must be considered to ensure consistent parsing behavior across real-world datasets.

Whitespace and encoding anomalies frequently interfere with column name detection. A robust system must remove leading and trailing whitespace and handle invisible characters such as the byte order mark (BOM) prior to applying **RegEx** matching. Columns with duplicate or missing headers should be automatically resolved by assigning fallback identifiers such as `column_1`, `column_2`, etc., to ensure that every field is addressable.

Special characters and multilingual content present additional complications. Matching patterns in names like "résumé" or "order#id" requires Unicode-aware **RegEx** and often normalization using forms such as NFKC to compare semantically equivalent strings. Ambiguities in naming patterns—such as fields ending in `_id`—may misleadingly suggest an integer type. In such cases, a tiered fallback strategy should apply: default to integer only if the content is strictly numeric, otherwise infer as string to prevent errors from hybrid or malformed inputs.

Columns where values exhibit mixed formats are particularly common in date fields. Even if the header indicates a date (e.g., contains "_date"), content-based validation must confirm consistent parsing across rows. A content parser should support flexible date formats (e.g., `YYYY-MM-DD`, `MM/DD/YYYY`, `June 4 2021`) and either coerce or flag rows that fail validation. Similarly, numeric strings with leading zeros, such as zip codes or identifiers, must not be misinterpreted as numbers. If the column name suggests identity (e.g., ends with `_id`), the presence of leading zeros should trigger a forced cast to string.

Handling missing or special "null" values—such as `null`, `N/A`, or empty strings—requires semantic understanding of their intent. Depending on the context, they may be dropped, cast to `None`, or retained as string representations. Proper delimiter handling is also essential. Rather than naively splitting on commas, systems must adhere to Request For Comments (**RFC**) 4180 or another specified CSV dialect, especially for fields that contain delimiters inside quoted strings.

Column name normalization is recommended to improve pattern matching reliability. This may involve replacing hyphens (-) with underscores (_) to align with standard naming conventions in downstream systems. Boolean inference should also account for a wide variety of lexical representations, including `1/0`, `true/false`, and `YES/NO`, and normalize them into a unified Boolean type, assuming no conflicting values are found.

All of these cases were part of the evaluation process to verify how well both the baseline heuristics and the **LLMs**-assisted inference generalize to structurally inconsistent or malformed CSV data. The edge-case evaluation confirms that successful schema inference requires more than just accurate model predictions—it must also include robust pre-processing, normalization, and error handling logic built into the parsing stack.

7 Conclusions

7.1 Template Generation

- regex logic sufficient to capture most relevant aspects of the CSV files
- 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
- 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

7.2 LLM Schema Inference

- high amount of computation capacities needed to run LLM locally
- LLMs are not yet able to detect the column name row in all cases
- a lot of hallucinating problems even with rather big models
- response times are not yet fast enough for real-time applications
- Model Dependency: Requires powerful language model to perform well
- Latency: Dependent on LLM processing time and local/hpc server speed (now not really a problem but was before)

- 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

test training model on schemapile in the future[Til24]

Bibliography

[1] S. L. S. C. A. W. Inacio Vieira Will Allred, "How Much Data is Enough Data? Fine-Tuning Large Language Models for In-House Translation: Performance Evaluation Across Multiple Dataset Sizes," *arXiv preprint arXiv:2409.03454*, 2024, doi: https://doi.org/10.48550/arXiv.2409.03454.

[Yua24] M. Z. X. H. L. D. S. H. D. Z. Yuan Sui Jiaru Zou, "TAP4LLM: Table Provider on Sampling, Augmenting, and Packing Semi-structured Data for Large Language Model Reasoning," *arXiv preprint arXiv:2405.14228*, 2024, doi: https://doi.org/10.48550/arXiv.2405.14228.

[Lei24] S. K. S. M. X. W.-P. C. T. K. T. K. T. A. Lei Liu So Hasegawa, "AutoDW: Automatic Data Wrangling Leveraging Large Language Models," *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, 2024, doi: https://doi.org/10.1145/3691620.3695267.

[Sah24] T. O. F. C. Sahar Iravani, "Towards More Effective Table-to-Text Generation: Assessing In-Context Learning and Self-Evaluation with Open-Source Models," *arXiv preprint*, no. 1, pp. 1–16, 2024, doi: https://arxiv.org/abs/2410.12878.

[tea23] M. A. team, "Mixtral of Experts: A High Quality Sparse Mixture-of-Experts," *Mistral AI Research Blog*, vol. 1, no. 1, p. 1, 2023, doi: https://mistral.ai/news/mixtral-of-experts.

[Cle25] T. W. Clement Farabet, "Introducing Gemma 3: The Most Capable Model You Can Run on a Single GPU or TPU," *Google Developers Blog*, vol. 1, no. 1, p. 1, 2025, doi: https://blog.google/technology/developers/gemma-3.

[Tea25] G. Team, "Gemma 3 Technical Report," *arXiv preprint arXiv:2503.19786*, 2025, doi: https://doi.org/10.48550/arXiv.2503.19786.

[Clo25] A. Cloud, "Qwen LLMs," *Alibaba Cloud Documentation*, Feb. 2025, [Online]. Available: https://www.alibabacloud.com/help/en/model-studio/what-is-qwen-llm

[Bil24] M. Bilenko, "New models added to the Phi-3 family, available on Microsoft Azure," *Microsoft Azure Blog*, May 2024, [Online]. Available: https://azure.microsoft.com/en-us/blog/new-models-added-to-the-phi-3-family-available-on-microsoft-azure/

[Dee25] DeepSeek-AI, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," *arXiv preprint arXiv:2501.12948*, Jan. 2025.

[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.

[Til24] M. H. S. S. Till Döhmen Radu Geacu, "SchemaPile: A Large Collection of Relational Database Schemas," *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, p. Article172, 2024, doi: https://doi.org/10.1145/3654975.

Acronyms

CPU	Central Processing Unit
DSL	domain-specific language
GPU	Graphics Processing Unit
GUI	Graphical User Interface
JSON	JavaScript Object Notation
LLM	Large Language Model
RFC	Request For Comments
RegEx	Regular Expressions

Bill Of Materials

Jayvee Template Generation

The template generation pipeline and associated testing scripts require the following software environment and dependencies:

- Python version: 3.8 or higher (tested with Python 3.10)
- Python packages:
  - pandas==2.0.3
  - jsonformer==0.12.0
  - jsonpatch==1.33
  - python-dateutil==2.8.2
  - typing-extensions==4.6.0

The tkinter library is used for GUI-based folder selection dialogs and is included as part of the standard Python distribution (version 3.8+).

Additionally, the pipeline requires the jv command-line tool, which must be installed and accessible in the system's PATH for executing .jv files and generating SQLite databases.

Notes

- The Python packages listed were installed with exact version pins to ensure reproducibility and compatibility during development and testing.
- The environment was tested on [your OS, e.g., Windows 10 / Ubuntu 22.04].
- The python_requires field in the setup configuration enforces minimum Python version requirements.