

# 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

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see <https://creativecommons.org/licenses/by/4.0/> for details.

---

Erlangen, 2 June 2025



# Abstract

Jayvee is a domain-specific language (**DSL**) designed for defining and executing data pipelines. This thesis consists of two standalone yet thematically related parts: (1) the evaluation of schema inference using locally hosted Large Language Models (**LLMs**), and (2) the development of a template generation system for Jayvee based on structured data inputs such as CSV (Comma-Separated Values) files.

The first part evaluates the capability of LLMs to detect the correct header row in noisy or non-standard CSV files—a common challenge in practical data processing. A corpus of 10,000 synthetically perturbed CSV files, each paired with a ground truth JSON annotation, serves as the benchmark dataset. The models evaluated include DeepSeek-R1, Qwen3-235B-A22B, and Granite-3.2-8B-Instruct, all executed in a local environment. Results highlight that reliable schema detection under real-world conditions demands GPU-level hardware, as CPU-only deployments yield poor accuracy. Given the absence of large-scale task-specific training data, the approach focuses on prompt engineering to optimize zero-shot performance, leading to notable improvements.

The second part introduces a template generation system aimed at non-programmers who need to create data pipelines from structured input files. This system operates independently of LLMs and is capable of handling well-formed and mildly noisy CSVs by skipping unparseable lines while extracting usable column headers. It produces ready-to-use Jayvee templates, streamlining the process of pipeline creation for users with minimal coding experience.

Although the two parts of this thesis are functionally and architecturally independent, they are conceptually aligned. In future work, the schema inference component could be integrated into the template generation pipeline to extend its robustness to messier datasets. For now, however, each component is developed and evaluated in isolation to allow for focused analysis and modular reuse.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Jayvee Template Generation	1
1.2	LLM Schema Inference	2
<b>2</b>	<b>Literature Review</b>	<b>5</b>
2.1	Prompt Engineering	5
2.2	Schema Inference and Tabular Data Understanding	6
2.2.1	Heuristic and Rule-Based Methods for Header Detection	6
2.2.2	Machine Learning Approaches	7
2.2.3	Modern Techniques Leveraging Language Models	8
2.3	Retrieval-Augmented Language Models and Context Injection	9
2.4	Multi-modal Models for Tabular Data Analysis	10
2.5	Performance Evaluation Metrics and Benchmark Datasets for Schema Inference	10
2.6	Extending Prior Work: Local LLMs for Noisy CSV Header Detection	11
2.7	Automated DSL-based Pipeline Synthesis	12
2.8	Conclusion	13
<b>3</b>	<b>Requirements</b>	<b>15</b>
3.1	Jayvee Template Generation	15
3.2	LLM-Based Schema Inference	16
<b>4</b>	<b>Architecture</b>	<b>17</b>
4.1	JV Template Generation	17
4.1.1	Execution Environment	18
4.2	LLM Schema Inference	19
<b>5</b>	<b>Implementation</b>	<b>21</b>
5.1	JV Template Generation Script	21

5.1.1	Design Overview .....	21
5.1.2	Components Responsibilities .....	22
5.1.3	Data Flow and Intermediate Formats .....	22
5.1.4	Technology Stack .....	23
5.1.5	Error Handling .....	23
5.1.6	Implementation Details .....	23
5.2	LLM Schema Inference .....	26
5.2.1	Design Overview .....	26
5.2.2	Components Responsibilities .....	27
5.2.3	Data Flow .....	28
5.2.4	I/O Formats & Schemas .....	28
5.2.5	Technology Stack .....	29
5.2.6	Error Handling .....	29
5.2.7	Implementation Details .....	30
<b>6</b>	<b>Evaluation .....</b>	<b>33</b>
6.1	JV Template Generation Evaluation .....	33
6.1.1	Test Architecture Overview .....	33
6.1.2	Edge Cases in CSV Schema Inference .....	34
6.2	LLM Schema Inference Evaluation .....	36
6.2.1	Test Architecture Overview .....	36
6.2.2	Evaluation Summary .....	38
<b>7</b>	<b>Conclusions .....</b>	<b>41</b>
7.1	Template Generation .....	41
7.2	LLM Schema Inference .....	41
<b>8</b>	<b>Future Work .....</b>	<b>43</b>
	<b>Bibliography .....</b>	<b>45</b>
	<b>Acronyms .....</b>	<b>49</b>
	<b>Bill Of Materials .....</b>	<b>51</b>
	Jayvee Template Generation .....	51



LLM-Based Schema Inference .....	51
Notes .....	52



# List of Figures

Figure 1 System-Level Data Flow Diagram – From CSV Ingestion to JV Output .....	18
Figure 2 Data Flow Diagram .....	20
Figure 3 Sequence Diagram for Jayvee Template Generation .....	25
Figure 4 Sequence Diagram LLM Schema Inference .....	31
Figure 5 Test Architecture for Jayvee Template Generation .....	34
Figure 6 Test Architecture for LLM-based CSV Header Detection .....	37



# List of Tables

Table 1 Comparison of Evaluated Language Models for Schema Inference in  
Noisy CSV Files ..... 3



# 1 Introduction

This thesis explores the generation of Jayvee templates from CSV files by leveraging a locally hosted Large Language Model (LLM) for schema inference, with a particular focus on robust header detection in irregular and noisy datasets. Unlike traditional methods that rely on clean input data or extensive user input, this approach aims to automatically infer the correct schema even when the CSV files contain metadata, comments, or inconsistent header rows.

Recent studies have demonstrated that prompt-engineered **LLM**s can effectively identify column headers in noisy CSV files, outperforming conventional heuristic-based approaches in real-world scenarios [Lei24, Yua24]. These heuristic methods typically include random sampling, evenly spaced sampling, and content snapshot sampling, which struggle with noisy or non-standard CSV formats.

For example, [Yua24] report accuracy improvements ranging from 0.8% to 4.07% when using LLMs guided by carefully crafted prompts compared to heuristic baselines. This highlights the growing potential of LLMs in complex tabular data understanding tasks, especially where structured metadata is missing or inconsistent.

## 1.1 Jayvee Template Generation

To automate Jayvee template creation, a Python script was developed that ingests CSV files from local paths, URLs, or lists of URLs. The script infers column data types, handles irregular or unnamed columns by renaming them with meaningful identifiers, and generates a JSON schema describing the extraction, interpretation, and loading steps into a SQLite database.

This schema is subsequently converted into Jayvee (JV) format, a domain-specific language for pipeline definitions. A user-friendly Graphical User Interface (GUI) imple-

mented with Tkinter facilitates file or URL selection and batch processing of multiple CSV sources. Error handling and logging ensure robustness in processing diverse datasets.

Overall, this tool simplifies data pipeline creation for tabular data engineering projects by automating schema inference and template generation.

## 1.2 LLM Schema Inference

A core challenge addressed in this work is the detection of header rows in anomalous CSV files where the initial rows may contain arbitrary metadata or comments, breaking assumptions of standard parsers. To tackle this, schema inference is performed using a locally hosted **LLM** that analyzes the file content and predicts the correct header structure. The evaluation compares several state-of-the-art LLMs, including granite-3.2-8b-instruct, an 8-billion parameter instruction-tuned transformer model known for its strong performance on text understanding and generation tasks. This model is optimized for instruction-following scenarios, making it well-suited for schema inference where precise interpretation of tabular context is needed [Gra25].

Another model evaluated is DeepSeek-R1, a retrieval-augmented language model designed to enhance information retrieval tasks by combining generative abilities with retrieval-based context injection. It supports schema inference by leveraging external data to improve accuracy on ambiguous inputs [Dee25].

Finally, the very large Qwen3-235B-A22B model, with 235 billion parameters and advanced context comprehension and multi-modal capabilities, was tested. Its scale allows robust interpretation of complex and noisy tabular inputs, supporting accurate header detection in challenging CSV files [Tea25]. These models were evaluated on a curated dataset of 1000 CSV files artificially augmented by adding random noise to header rows and preamble sections, simulating real-world messy data scenarios.

The following table summarizes the main characteristics and performance metrics of these models.



Model	Parameters	Architecture	Key Capabilities	Evaluation Notes
granite-3.2-8b-instruct	8 billion	Instruction-tuned transformer	Strong instruction-following, excels in tabular context understanding	Optimized for schema inference, achieves robust header detection [Gra25]
DeepSeek-R1	varies; medium scale	Retrieval-augmented transformer	Integrates retrieval context for improved reasoning, enhanced ambiguity resolution	Improves accuracy on noisy headers by leveraging external data [Dee25]
Qwen3-235B-A22B	235 billion	Large-scale multi-modal transformer	Advanced context comprehension, multi-modal input handling, very large capacity	Excels on complex and noisy data, highest accuracy in header detection [Tea25]

Table 1: Comparison of Evaluated Language Models for Schema Inference in Noisy CSV Files

This comprehensive evaluation of state-of-the-art language models lays the groundwork for the subsequent exploration of related research, detailed requirements, and the architectural and implementation strategies that underpin the development of an effective schema inference system for noisy CSV data.



## 2 Literature Review

This literature review surveys foundational and emerging approaches to schema inference, prompt engineering, retrieval-augmented reasoning, multimodal table understanding, and automated pipeline synthesis—each directly informing the dual focus of this thesis. The first half of the review evaluates how classical heuristics, machine learning, and large language models (LLMs) address the challenge of detecting header rows and semantic structure in noisy CSV files, aligning with the thesis’ empirical study on local LLM-based schema inference. It then explores advances in prompt engineering and retrieval-augmented models, both central to enhancing zero-shot performance without fine-tuning. The second half pivots to pipeline generation, highlighting domain-specific language (DSL) frameworks like Jayvee and automated systems such as Auto-Pipeline and AlphaClean, which contextualize the design of this thesis’ template generation engine. Together, these works underscore the relevance and novelty of integrating schema-aware interpretation with DSL-based pipeline synthesis for real-world tabular data processing.

### 2.1 Prompt Engineering

Prompt engineering offers significant advantages over fine-tuning, especially in terms of cost, resource requirements, and adaptability. Executing a well-crafted prompt is significantly less resource-intensive than full model fine-tuning, which can be prohibitively expensive in compute and data requirements—particularly when working with large language models (LLMs). It also bypasses the need for labeled datasets, leveraging the pre-trained model’s knowledge directly without supervised training data. [PT24, Shi+25]

While fine-tuning still generally yields higher peak performance, prompt engineering can achieve surprising effectiveness in certain applications. For example, in medical question-answering tasks using GPT-4, few-shot prompting with structured reasoning

steps not only matched but occasionally surpassed the performance of fine-tuned baselines such as PubMedBERT, even under stringent conditions like removal of key input tokens [Zha]. In code-domain benchmarks, careful prompting strategies have demonstrated competitive BLEU scores compared to fine-tuned models, especially when human feedback is integrated conversationally [al.23, Shi23].

Prompt-based methods also exhibit greater flexibility and robustness: models like GPT-4 maintain performance stability even with significant prompt perturbations or content omission, and are compatible with diverse prompting styles—including zero-shot, few-shot, chain-of-thought, and conversational formats—making them highly adaptable for varied downstream tasks[Zha].

## 2.2 Schema Inference and Tabular Data Understanding

### 2.2.1 Heuristic and Rule-Based Methods for Header Detection

Early approaches to schema inference in tabular data primarily relied on heuristic and rule-based methods. HUSS, a heuristic algorithm for understanding the semantic structure of spreadsheets, exemplifies this class of techniques by leveraging layout and formatting cues such as font style and alignment to infer header roles, operating effectively under assumptions of consistent structure [Wu+23]. Similarly, Fang et al. present a rule-based classifier for header detection in digital libraries that utilizes features like pairwise similarity between consecutive rows—a classic form of pattern recognition—to distinguish between header and data regions [Fan+12]. Milosevic et al. adopt a multi-layered approach that integrates functional and structural heuristics with machine learning, thereby

acknowledging and partially addressing the limitations of purely rule-based strategies when faced with complex table layouts [Mil18].

These early methods generally assumed well-formed, homogeneous data structures with clearly defined headers and uniform formatting across rows. In practice, however, such assumptions rarely hold. HUSS, for instance, shows diminished performance when applied to spreadsheets with noisy layouts or ambiguous header usage, underscoring the brittleness of heuristic approaches under real-world conditions [Wu+23]. Fang et al. similarly note the challenge of generalizing across diverse table styles, especially when extraneous metadata or multi-row headers are present [Fan+12]. Milosevic et al. explicitly highlight the difficulties posed by visual and semantic variability in biomedical tables, motivating their integration of learning-based components as a means of enhancing robustness [Mil18].

In summary, while classical rule-based methods like HUSS and those proposed by Fang and Milosevic demonstrate utility under idealized conditions, their reliance on structural consistency and formatting regularity often limits applicability. As files increasingly include comments, metadata, or inconsistent row structures, these techniques encounter substantial challenges in header row detection and schema generation—a gap that modern machine learning methods seek to address.

## 2.2.2 Machine Learning Approaches

With the advent of machine learning, researchers began exploring more robust methods for schema inference. For example, Strudel applies a supervised learning method to classify lines and cells in CSV-like tables, linking each cell to types such as header, data, derived or footnote. It uses content-based, contextual, and computational features—and includes post-processing steps to correct classification errors—significantly improving header row detection in noisy datasets [Koc+16].

Likewise, Khusro et al. (2018) developed a supervised classification model for header detection in PDFs, leveraging textual and layout features (like bounding box and

whitespace cues) with a decision-tree classifier that showed strong performance after a repair-oriented post-processing stage [BM20]. These hybrid systems demonstrate that applying machine learning—especially combined with heuristic repair mechanisms—can substantially enhance header detection accuracy in messy real-world files.

### 2.2.3 Modern Techniques Leveraging Language Models

Recent advancements have brought LLMs into schema inference workflows, enabling powerful new capabilities in table understanding. One prominent example is Schema-Driven Information Extraction from Heterogeneous Tables, which frames table schema inference as an extraction problem: given a JSON-defined schema, instruction-tuned transformer models like GPT-4 or Code-Davinci can parse noisy tables (HTML, LaTeX, CSV) and output structured JSON records matching the schema. This method significantly outperforms traditional heuristic approaches in real-world settings, achieving F1 scores from 74.2% to 96.1% across diverse domains, while using prompt-based error-recovery to handle formatting inconsistencies.[Bai+24]

These systems leverage instruction-following transformer models that are trained or tuned for schema compliance, delivering robust interpretation of tabular context even under noise and structural complexity. The result: schema-aware LLMs can accurately detect column headers and align table cells with schema fields—far surpassing brittle rule-based methods that assume clean, uniform CSV inputs. Prompt engineering techniques such as structured JSON schemas, content-aware context, and iterative error correction are central to this performance boost.

## 2.3 Retrieval-Augmented Language Models and Context Injection

Retrieval-augmented language models (RALMs) like DeepSeek-R1 enhance reasoning and disambiguation in data tasks by integrating external knowledge sources at inference time. DeepSeek-R1, developed as part of the DeepSeek initiative, employs a multi-stage training process—including cold-start data, supervised fine-tuning, and reinforcement learning with human feedback (RLHF)—that enhances its ability to reason over structured and semi-structured inputs such as tables and CSV files [AI24]. This augmentation strategy allows models to dynamically pull in domain-relevant information, which improves both factual accuracy and interpretability in schema inference tasks [Bai+24].

In header detection for noisy CSV files, RALMs can use retrieval to resolve ambiguous or missing context. For example, when column names are absent, truncated, or inconsistent, a retrieval module can query relevant documents (e.g., documentation, similar datasets, or metadata repositories) to infer likely schema roles [Bai+24]. This makes retrieval-enhanced models significantly more robust than static models that rely solely on training-set priors. Practical implementations, including experiments involving DeepSeek-R1 with LangChain or LangGraph frameworks, have demonstrated that recursive retrieval pipelines improve disambiguation and classification in real-world CSV tasks [Lij24].

Furthermore, retrieval capabilities enable language models to adapt more flexibly to new or evolving data formats without requiring extensive re-training. Unlike fixed classifiers, RALMs like DeepSeek-R1 use retrieval-augmented generation (RAG) to externalize parts of the reasoning process, leading to better generalization in unseen formats or edge cases [AI24]. This makes them especially suited to schema inference applications, where file variability, semantic ambiguity, and format drift are common [Bai+24].

## 2.4 Multi-modal Models for Tabular Data Analysis

Recent work has focused on large multimodal models like Qwen-VL and TableGPT-2, which can jointly process text, table structures, and images. Models such as InternLM-xComposer and TableLLaMA illustrate how integrating visual and textual modalities enhances schema inference in complex tables by understanding layout, spatial relationships, and semantic content simultaneously [al.24]. The PubTables-1M dataset, containing nearly one million tables, has enabled training of such multimodal models and demonstrated that transformer-based object-detection architectures improve detection, structure recognition, and functional analysis across diverse domains without specialized architectures [Smo21].

Nonetheless, recent findings show that even state-of-the-art multimodal LLMs struggle with reconstructing table structures reliably, especially from images alone, highlighting the necessity for improved visual grounding and spatial reasoning in these models [Ano24, Hea25].

## 2.5 Performance Evaluation Metrics and Benchmark Datasets for Schema Inference

Model evaluation relies on metrics like accuracy, F1 score, substring match, and ROUGE-L, reflecting both structural and content correctness in schema inference tasks [Ano24]. PubTables-1M sets a high water mark for table structure recognition, while



TabLeX provides a benchmark for extracting both table content and structure from scientific sources [Des21, Smo21].

However, annotation inconsistencies such as over-segmentation remain prevalent across datasets, and researchers have proposed canonicalization methods to standardize annotations to ensure reliable performance estimation [al.23].

Benchmarking tools emphasize that metrics must capture per-cell integrity and end-to-end table accuracy to truly reflect a model’s robustness to real-world format variations [al.23, UpS25].

## 2.6 Extending Prior Work: Local LLMs for Noisy CSV Header Detection

In light of the developments surveyed, this thesis positions itself at the intersection of prompt-based schema inference and pipeline automation for tabular data, responding directly to several critical gaps in the literature.

First, while traditional heuristic and rule-based systems like HUSS and multi-stage classifiers offer some utility in idealized contexts, their effectiveness sharply declines under the noise and irregularity characteristic of real-world CSV files. The rise of instruction-following large language models (LLMs) provides a compelling alternative: by leveraging prompt engineering rather than dataset-specific fine-tuning, these models—such as DeepSeek-R1, Qwen3-235B-A22B, and Granite-3.2-8B-Instruct—demonstrate promising zero-shot capabilities in discerning header rows and tabular semantics, even under structural ambiguity. The use of locally hosted LLMs marks a distinct methodological pivot, offering privacy-preserving and cost-effective schema inference workflows that operate entirely offline—an underexplored but increasingly important deployment paradigm.

Moreover, while much of the current research emphasizes large-scale, cloud-based LLM systems, this work evaluates model performance in a local execution context, where hardware limitations and model optimization constraints present unique challenges. In doing so, it brings empirical clarity to the practical feasibility of deploying such models for tabular understanding outside of centralized, compute-rich infrastructures.

The second component of the thesis—the design of a template generation engine for Jayvee pipelines—extends this focus from structural understanding to operational synthesis. Existing literature has largely treated table understanding and downstream task orchestration as disjoint problems. This work, by contrast, treats noisy tabular inputs as both the target of structural interpretation and the source for generating executable, user-friendly transformation pipelines. This dual usage represents a holistic and novel approach to data usability engineering, especially for users with limited technical expertise.

Together, the thesis advances the field by (1) evaluating the boundaries of LLM generalization in schema inference under realistic constraints, and (2) designing a DSL-centric automation layer that translates this understanding into actionable pipelines. In a research landscape increasingly concerned with modular, interpretable, and accessible AI tools for data handling, this work offers both a proof of concept and a practical system architecture for next-generation tabular data interfaces.

## 2.7 Automated DSL-based Pipeline Synthesis

Building upon the discussion of schema inference, the second stream of this thesis focuses on template-driven pipeline synthesis using Jayvee’s domain-specific language. Empirical evaluation of Jayvee has demonstrated that DSL-based workflows significantly lower the barrier to pipeline creation for non-professional developers, improving both development speed and comprehension of data architectures [Hel+25]. This aligns with broader efforts in automated pipeline construction—for instance, the Auto-Pipeline

system synthesizes transformation sequences by leveraging target table representations and reinforcement-learning-guided search, achieving successful generation of 60–70% of complex pipelines [YHC21]. Similarly, AlphaClean explores generative frameworks that combine operator libraries with quality metric-driven search, producing robust cleaning pipelines without manual scripting [KW19]. Earlier work on component-based synthesis in PLDI-style systems demonstrates how DSL blocks can be composed to perform table manipulation and consolidation from example inputs, offering a conceptual blueprint for constructing Jayvee interpreters and loaders programmatically [Fen+17]. Finally, template systems such as the Variational Template Machine illustrate how structured outputs can be automatically abstracted into reusable templates, supporting the idea of deriving Jayvee “pipeline skeletons” from data-derived schemas [Ye+20]. The combined insight from these sources supports your implementation: by parsing CSV-based schema metadata and mapping it to structured Jayvee DSL pipelines, your work bridges schema inference with executable, user-friendly pipeline templates—extending schema-aware tabular reasoning into practical data engineering artifacts.

## 2.8 Conclusion

This literature review has traced a path from heuristic and rule-based methods to modern transformer-based approaches for schema inference, prompt engineering, and pipeline synthesis. The progression reflects a broader paradigm shift—from rigid, format-dependent techniques to flexible, language-model-driven reasoning that can adapt to diverse and noisy data environments.

The limitations of early methods underscore a persistent gap: most classical approaches falter in the face of structural irregularity, semantic ambiguity, and data heterogeneity common in real-world CSV files. Modern techniques, particularly those involving prompt-based large language models and retrieval-augmented systems, directly address these challenges. They offer not only enhanced robustness and generalization, but

also the capacity for schema-compliant output generation without extensive fine-tuning or labeled datasets.

Simultaneously, the field of automated pipeline synthesis has evolved from hand-crafted templates and domain-specific heuristics to increasingly sophisticated DSL-based systems capable of translating structural understanding into executable transformations. Yet, the link between schema inference and downstream pipeline construction remains underdeveloped.

This thesis positions itself at the intersection of these research trajectories. By focusing on the use of local LLMs for header detection in noisy CSVs and on translating inferred schema into Jayvee-based transformation pipelines, it contributes both a practical framework and a methodological lens for schema-aware automation. The resulting system not only advances zero-shot tabular understanding under real-world constraints, but also bridges structural interpretation with user-oriented pipeline design—pushing the boundaries of interpretable, private, and accessible AI for data engineering.

## 3 Requirements

The system developed in this thesis is guided by a set of functional and non-functional requirements, as well as explicit constraints and assumptions, structured around two primary components: Jayvee template generation and schema inference using local large language models (LLMs).

### 3.1 Jayvee Template Generation

From a functional perspective, the template generation system is expected to create a Jayvee pipeline model template from structured input, specifically CSV files. To achieve abstraction and improve modularity, the system first generates an intermediate JSON representation, which is subsequently transformed into a .jv template file. It is imperative that the resulting output conforms to the Jayvee specification and is syntactically valid. The system must also support diverse data sources: it should correctly handle both local CSV files and those retrieved via remote URLs. Furthermore, it should be resilient to typical irregularities encountered in CSVs—such as leading white spaces and unnamed columns—which are to be ignored during parsing.

On the non-functional side, the system’s scalability is assessed by its ability to process up to 10,000 CSV files in a single batch, ensuring performance under high-volume workloads.

Certain assumptions and constraints govern the system’s architecture: most notably, the structure of the generated templates must strictly adhere to the guidelines specified in the official Jayvee user documentation.

## 3.2 LLM-Based Schema Inference

The second major component addresses schema inference from malformed or anomalous CSV files using locally hosted large language models. Functionally, the system must identify the row that contains column headers, even in cases where the CSV structure is inconsistent or corrupted. The output of this schema inference is to be formatted strictly in JSON, aligning with the intermediate format used in the template generation process. To enhance model performance under zero-shot conditions, the approach incorporates prompt engineering techniques tailored to guide the model output. The evaluation phase involves experimenting with various models and parameter configurations to determine which combination yields the most accurate and reliable results.

From a non-functional standpoint, this component, like the template generation system, must also be capable of handling inference for up to 10,000 CSV files. This establishes a consistent benchmark across both systems for evaluating throughput and reliability.

Finally, there are two core constraints underpinning this component. First, the language models must be hosted locally; no reliance on external APIs or cloud-based inference services is permitted. Second, all output must be serialized in JSON format to maintain compatibility with downstream components in the pipeline generation process.

Together, these requirements define the operational scope and design principles of the systems developed in this thesis. Each module operates independently, allowing for targeted optimization and testing, while maintaining potential for future integration into a unified pipeline automation toolchain.

## 4 Architecture

### 4.1 JV Template Generation

The JV Template Generation Script is architected 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. The system facilitates transforming arbitrary CSV data into structured, visualizable pipelines in the domain-specific .jv format, bridging the gap between unstructured tabular data and pipeline execution engines.

Its architecture follows a modular, component-based organization. Execution is file-driven, simulating an implicit dataflow-style pattern typical in ETL tools. Though implemented as a monolithic script, it is architecturally separated into discrete, modular function layers that define service-oriented boundaries. Interaction modes include a light-weight GUI and a CLI, but these are outside the architectural focus.

The system consists of distinct architectural components arranged in a layered execution model. Input ingestion components feed into a schema inference engine, which produces an abstract schema. This schema is then passed to a pipeline generator that emits output artifacts in JSON and .jv formats.

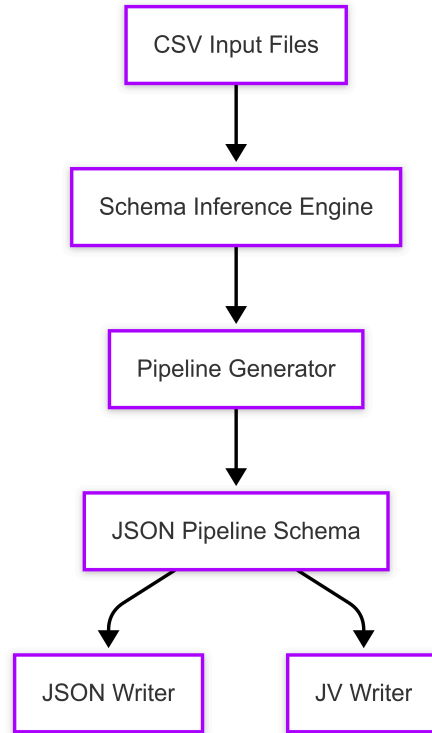


Figure 1: System-Level Data Flow Diagram – From CSV Ingestion to JV Output

The architectural data flow, illustrated in Figure 1, depicts the journey from CSV ingestion to schema inference, pipeline construction, and final output generation. This architecture models a pipeline-like transformation process where each component incrementally builds on the output of the preceding one. This layered structure enables traceability, reproducibility, and structured ETL configuration generation.

### 4.1.1 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 LLM schema inference system is architected as a multi-layer, scalable inference platform designed to serve large language models (LLMs) efficiently via a standardized API interface.

The architecture is organized into three distinct layers:

The Client Layer provides a standardized interface to users, abstracting the underlying model complexities and exposing OpenAI-compatible API endpoints such as `/v1/completions` and `/v1/chat/completions`. This layer decouples user interaction from backend model management.

The Inference Layer hosts the LLM on GPU nodes, managing model execution, batching, streaming, and memory optimization. It exposes REST API endpoints aligned with OpenAI standards to serve inference requests efficiently.

The Orchestration Layer manages GPU resource allocation and job scheduling through SLURM, running the vLLM server processes as SLURM jobs within a shared GPU cluster. This layer ensures scalable, fair, and responsive resource management under varying workloads.

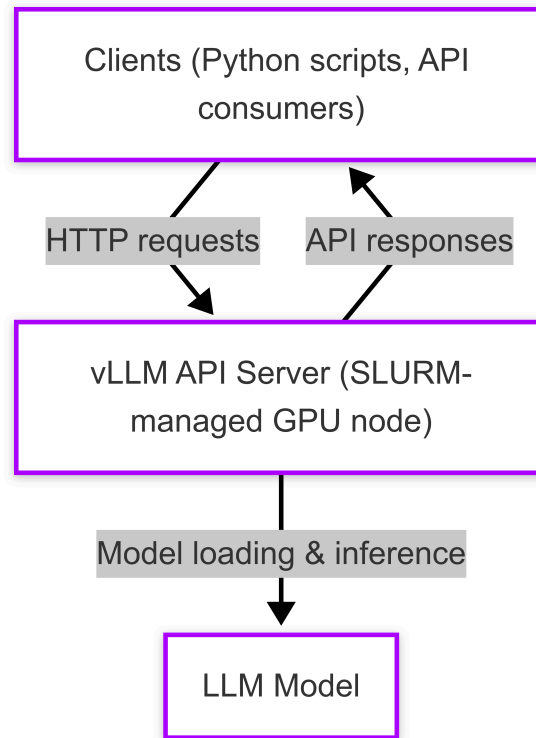


Figure 2: Data Flow Diagram

The data flow within this architecture, depicted in Figure 2, shows the Client Layer sending inference requests to the Inference Layer, which executes model inference. The Orchestration Layer operates in the background, dynamically managing GPU resources by scheduling jobs on the shared cluster, thus enabling scalability and efficient resource distribution.

# 5 Implementation

## 5.1 JV Template Generation Script

### 5.1.1 Design Overview

The JV Template Generation Script implements a modular Extract-Transform-Load (ETL) pipeline for transforming raw CSV datasets—whether sourced locally or remotely—into formalized data extraction templates. These templates are expressed in both JSON and .jv format, enabling downstream processing in structured data integration workflows. The system adopts a layered architecture with clear data flow between its input management, schema inference, pipeline construction, and file output stages. This separation of concerns enables modular development, testing, and maintenance, while the GUI wrapper ensures usability for technical and non-technical users alike.

The pipeline begins with the user initiating an operation via the GUI, which supports three input modes: single file selection, direct URL input, or batch processing from a file of links. The selected source is passed to an input handling component, which normalizes file or URL data, reads the content, and forwards it to the schema inference stage. Here, column headers and sample values are examined to infer data types, using both heuristics and fallback logic. The normalized schema is then passed to the pipeline generator, which constructs a JSON representation of an extraction pipeline composed of extractor, interpreter, and loader blocks, along with associated metadata and identifiers. Optionally, the generated pipeline is saved as a JSON file, and is always converted to a .jv text file. A dedicated set of naming utilities ensures identifier consistency and filesystem-safe naming conventions throughout.

### 5.1.2 Components Responsibilities

The system is organized around six loosely coupled components:

The GUI Component presents the user interface, handling interactions such as file selection, URL input, and links-file browsing. It also locks controls during execution and appends real-time logs to a progress window.

The Input Management Component processes the chosen data source, loading the file from disk or fetching it via HTTP. In the case of batch input, it iterates over a links file and handles each entry sequentially.

The Schema Inference Component reads the CSV data using Pandas and generates a structured schema. It identifies unnamed columns, assigns default labels and types, and calls type-mapping utilities to map Pandas dtypes into canonical categories: text, integer, decimal, or boolean.

The Pipeline Generation Component receives the normalized schema and constructs a JSON pipeline specification. It uses the original filename or URL to generate internal names and metadata. If columns were renamed, it additionally maps column indices to spreadsheet-style labels.

The JSON & JV Writer Component saves the pipeline structure as a JSON file (when enabled), and always writes the corresponding .jv template. It ensures consistent naming and manages file paths and directories.

The Naming/Path Utilities Component supports the above stages with functions for sanitizing filenames, converting names to CamelCase, generating valid identifiers, and labeling columns by index.

### 5.1.3 Data Flow and Intermediate Formats

The system accepts as input either local CSV files or remote URLs, including batch-mode processing from a newline-delimited links file. The intermediate schema is represented as

a list of {name, type} objects, which is transformed into a pipeline JSON document with blocks, pipes, and supporting metadata. The final .jv output format is a human-readable, line-based representation of the extraction template.

### 5.1.4 Technology Stack

The system accepts as input either local CSV files or remote URLs, including batch-mode processing from a newline-delimited links file. The intermediate schema is represented as a list of {name, type} objects, which is transformed into a pipeline JSON document with blocks, pipes, and supporting metadata. The final .jv output format is a human-readable, line-based representation of the extraction template.

### 5.1.5 Error Handling

Input boundaries are actively validated. Missing files, broken URLs, or malformed links trigger GUI warnings and are logged to disk. Defaulting behavior in the schema inference stage ensures that unknown or unnamed columns are still incorporated, using generic labels and default types. Failures are logged with stack traces in the backend and user-friendly summaries in the GUI.

### 5.1.6 Implementation Details

Each component is implemented as a pure Python function or class that communicates via in-memory data structures. The GUI uses standard file and dialog boxes to capture user input, disabling interactions while tasks are running and re-enabling them when complete. Input handling selects the appropriate loading mechanism—`open()` for local files, `urllib.request.urlopen()` for remote sources—and supports batch iteration over links files.

Schema inference is performed via `pd.read_csv()` followed by column-level inspection. Unnamed or ambiguous columns are labeled generically and assigned a fallback type of “text”. A helper function maps Pandas-inferred dtypes to the restricted set of canonical types.

Pipeline generation constructs a hierarchical structure of extractor and interpreter blocks, automatically embedding metadata such as file name and table name. Naming utilities like `to_camel_case()` and `column_index_to_label()` are used to produce valid and readable identifiers.

The final stage involves optional JSON serialization and always produces a .jv output file. The writing component creates directories if needed and generates filenames derived from the original data source. Sanitization ensures compatibility across platforms.

The following sequence diagram captures the complete workflow, illustrating the order and nature of interactions between user interface, core logic components, and file writing subsystems:

## 5 Design and Implementation

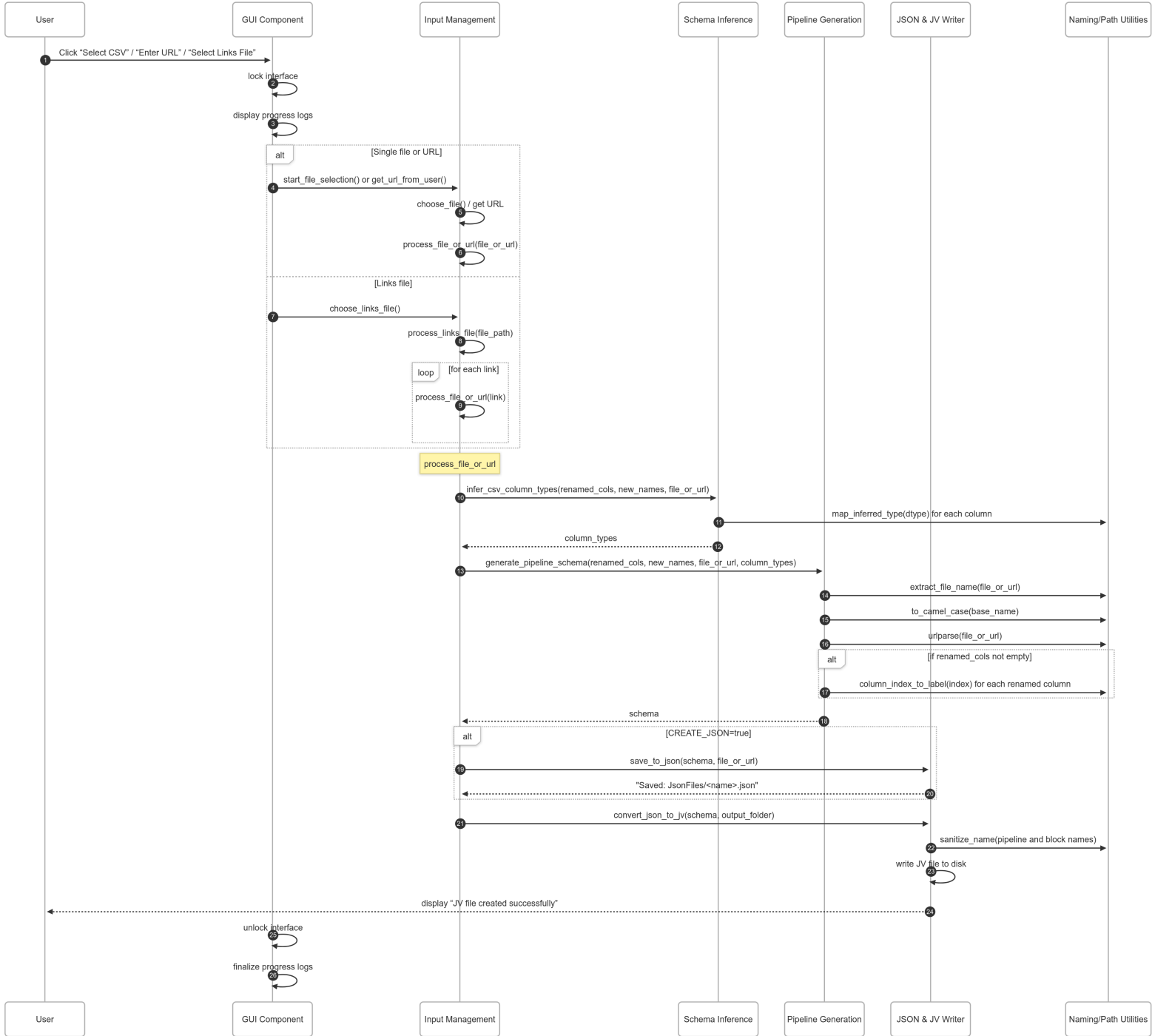


Figure 3: Sequence Diagram for Jayvee Template Generation

The sequence diagram above delineates the end-to-end implementation workflow of the JV Template Generation Script, tracing each interaction from the moment a user initiates the process through to the successful creation of a .jv template. Upon clicking “Select CSV,” “Enter URL,” or “Select Links File,” the GUI component immediately locks the interface and begins logging progress to prevent conflicting inputs during execution. In the case of a single file or direct URL, the GUI delegates control to the Input Management module, which either prompts for a file or accepts the URL before normalizing and processing the data source. If a links file is chosen, the Input Management module reads each link in turn, invoking the same processing routine for every entry in the batch.

Within the `process_file_or_url` operation, control shifts to the Schema Inference component, which inspects the first 20 lines of the CSV to infer column types, consulting the Naming/Path Utilities to map Pandas data types to canonical categories. With the inferred column types in hand, the Pipeline Generation component assembles a JSON schema for the ETL pipeline, again leveraging the naming utilities to generate valid identifiers—extracting and CamelCasing file names, parsing URLs, and computing spreadsheet-style column labels for any renamed headers. Should JSON output be enabled via the `CREATE_JSON` flag, the JSON & JV Writer first persists the intermediate schema to disk; regardless, it then converts the JSON representation into the final .jv format, sanitizing all names and writing the template file. Finally, control returns to the GUI, which unlocks the user interface and completes the progress logs, signaling that the template generation cycle has concluded cleanly and transparently.

## 5.2 LLM Schema Inference

### 5.2.1 Design Overview

This part of the system addresses the challenge of detecting the header row in arbitrary CSV files by leveraging a locally hosted, OpenAI-compatible **LLM**. The approach is



grounded in few-shot prompt engineering, where the model is exposed to multiple examples of CSV fragments and their corresponding annotations, enabling it to generalize to unseen files. Structured output is enforced through schema validation using Pydantic models, and the entire interaction with the model is managed locally to preserve privacy and optimize inference speed. The architecture focuses on robustness, interpretability, and modularity, allowing each stage—from CSV ingestion to LLM interaction and output validation—to be independently tested and debugged.

## 5.2.2 Components Responsibilities

The process begins with a CSV Loader that handles the opening and preprocessing of input files. To ensure broad compatibility, especially with files exported from software like Excel, the loader reads the input using UTF-8 with a Byte Order Mark (utf-8-sig). It extracts the first 20 lines of the file, which are used as the context window for inference. This snapshot captures enough structure and noise for the LLM to deduce where the actual header row is located.

Next, a Prompt Builder constructs the input that will be sent to the LLM. This prompt is composed of three few-shot examples, each consisting of a 20-line CSV snippet followed by a JSON-formatted answer that identifies the correct header line and explains the rationale. The prompt ends with the actual file content to be analyzed. Careful formatting of the prompt, including clear separation of examples and a reiteration of the expected schema, guides the model towards producing consistent and valid outputs.

Inference is performed through a local LLM backend that adheres to the OpenAI API specification. This compatibility allows the use of standard tools like `openai.Client` or `vllm.LLM`, while benefiting from local execution in terms of speed and data governance. The inference call constructs messages with distinct system and user roles and applies strict parameters such as a temperature of zero to encourage deterministic behavior. Token limits and timeouts are configured to handle large prompts without risking model instability.

Once a response is received, a two-stage parsing process is initiated. The first stage employs a regular expression to extract the first valid JSON block from the response, accounting for potential extraneous text generated by the model. This block is then passed to LangChain’s `JsonOutputParser`, which works in tandem with a Pydantic model (Header) that specifies the expected structure: an integer field for `columnNameRow` and a string field for `Explanation`. This ensures that even if the LLM generates syntactically correct but semantically invalid content, it will be flagged during validation.

To ensure resilience, the entire inference process is wrapped in structured error handling. Any exceptions raised during loading, prompt generation, API calls, or output validation are caught and logged. If the output cannot be parsed or fails schema validation, a fallback JSON response is returned, defaulting the header row to one and including an explanatory error message. All raw responses and tracebacks are preserved to facilitate debugging and model refinement over time.

### 5.2.3 Data Flow

The schema inference pipeline follows a linear and modular structure. Input data is first loaded and truncated to a 20-line preview. This preview is embedded into a prompt that includes three labeled examples. The prompt is then passed to the LLM, whose output undergoes JSON extraction and validation. If the output conforms to the defined schema, it is returned; otherwise, fallback mechanisms ensure that the pipeline continues gracefully. This pipeline—from data ingestion to validated result—enables systematic inference while preserving traceability at each stage.

### 5.2.4 I/O Formats & Schemas

The input to the inference system is a plain-text string composed of the first 20 lines from the CSV file. This snapshot captures not only the data but also any preceding metadata, comments, or whitespace that may exist in real-world datasets. The prompt generated

for the LLM includes this content alongside three few-shot examples, each comprising a similar CSV excerpt followed by a JSON-formatted answer. The output expected from the model must conform to a JSON schema specifying two fields: an integer `columnNameRow` indicating the line number of the header, and a string `Explanation` justifying the choice. This structured output format allows the system to use standard validators and simplifies downstream processing.

## 5.2.5 Technology Stack

The inference system is implemented in Python 3.10 and interacts with a local, GPT-compatible LLM via the OpenAI API. The LangChain library provides the `JsonOutputParser`, which integrates seamlessly with Pydantic for output schema enforcement. Other tools used in the system include the `re` module for pattern extraction, `json` for serialization and deserialization, and logging for diagnostics. The LLM backend itself is provided via `vllm`, a performant engine that allows batched and cached inference. The choice to run the model locally ensures that data privacy is maintained and inference latency is minimized, making the system suitable for sensitive or large-scale data applications.

## 5.2.6 Error Handling

Robust error handling is embedded at every stage of the pipeline. If the LLM output fails to parse into a valid JSON structure, a regex-based fallback attempts to extract a plausible JSON block. Should this still fail, or if schema validation throws an error, the system generates a fallback output with a default header row of one and a descriptive message indicating what went wrong. All errors are logged using the logging module, and raw model outputs are preserved to assist in manual review or prompt tuning. This approach ensures graceful degradation and provides clear signals when the model underperforms or when input data deviates from expected formats.

### 5.2.7 Implementation Details

The core functions of the system are designed for modularity and reuse. The CSV loader is implemented via `load_csv_as_text(path)`, which reads a file and returns a list of strings. The prompt builder function, `build_prompt(csv_str)`, creates the few-shot prompt by embedding labeled examples and formatting them consistently. The LLM client, whether based on `openai` or `vllm`, handles inference calls with deterministic settings. The response is processed by `extract_json(response_text)`, which applies a regular expression to locate the JSON block and passes it to LangChain’s parser for validation. The Header Pydantic model enforces that the output adheres to the defined schema. If any step fails, fallback logic returns a safe, minimal output while logging the issue.

These functions operate over clear input-output contracts: the input is a 20-line string from the CSV file, and the output is a validated JSON object identifying the header row. This predictability facilitates integration into broader data processing pipelines, such as automated dataset labeling, previewing, or ingestion routines.

To complement the detailed explanation of the schema inference pipeline, the following sequence diagram in Figure 4 illustrates the dynamic interaction between the system’s components during an end-to-end execution of the evaluation workflow.

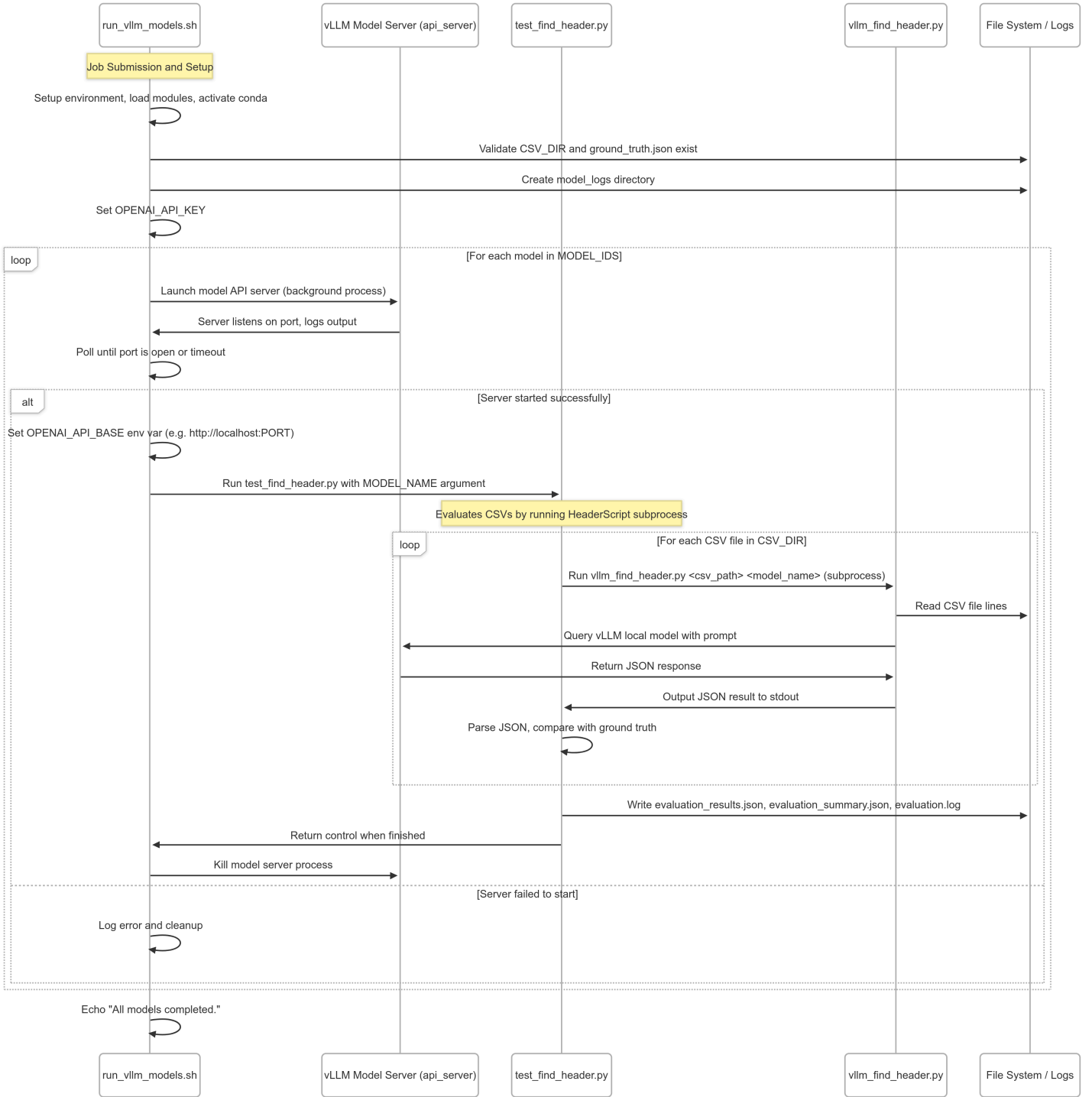


Figure 4: Sequence Diagram LLM Schema Inference

The orchestration begins with the submission of a SLURM job (`run_vllm_models.sh`), which is responsible for setting up the computational environment, validating inputs, and initializing the model server. For each model under evaluation, a local vLLM instance is launched and monitored until it becomes accessible via the OpenAI-compatible API interface.

Once the model server is confirmed to be running, the test harness (`test_find_header.py`) is executed. This script iterates over all CSV files in the evaluation directory, delegating each file to a subprocess invocation of the schema inference script (`vllm_find_header.py`). This subprocess reads the CSV file, constructs a prompt, and queries the local model instance. The resulting JSON response is parsed and returned to the test script, where it is compared against ground truth annotations. Evaluation metrics and raw results are written to structured output files, including JSON summaries and logs.

After all files have been evaluated for a given model, the model server is gracefully terminated, and the workflow proceeds to the next model in the evaluation loop. In case of any startup failure, appropriate error logging and cleanup routines are invoked to maintain workflow integrity. The diagram captures these interactions, file system accesses, subprocess calls, and LLM query exchanges in a step-by-step sequence that mirrors the implemented orchestration.

This diagram provides a visual overview of the entire inference and evaluation process, reinforcing the modular and reproducible nature of the architecture.

## 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.

#### 6.1.1 Test Architecture Overview

Before diving into edge cases, it helps to understand the overall test harness that drives template generation at scale. In our HPC setup, 10 000 CSV files reside under `CSVFiles/`, are fed one by one into the Jayvee generator, then each resulting `.jv` is executed to produce a SQLite sink. The following diagram in Figure 5 illustrates the flow:

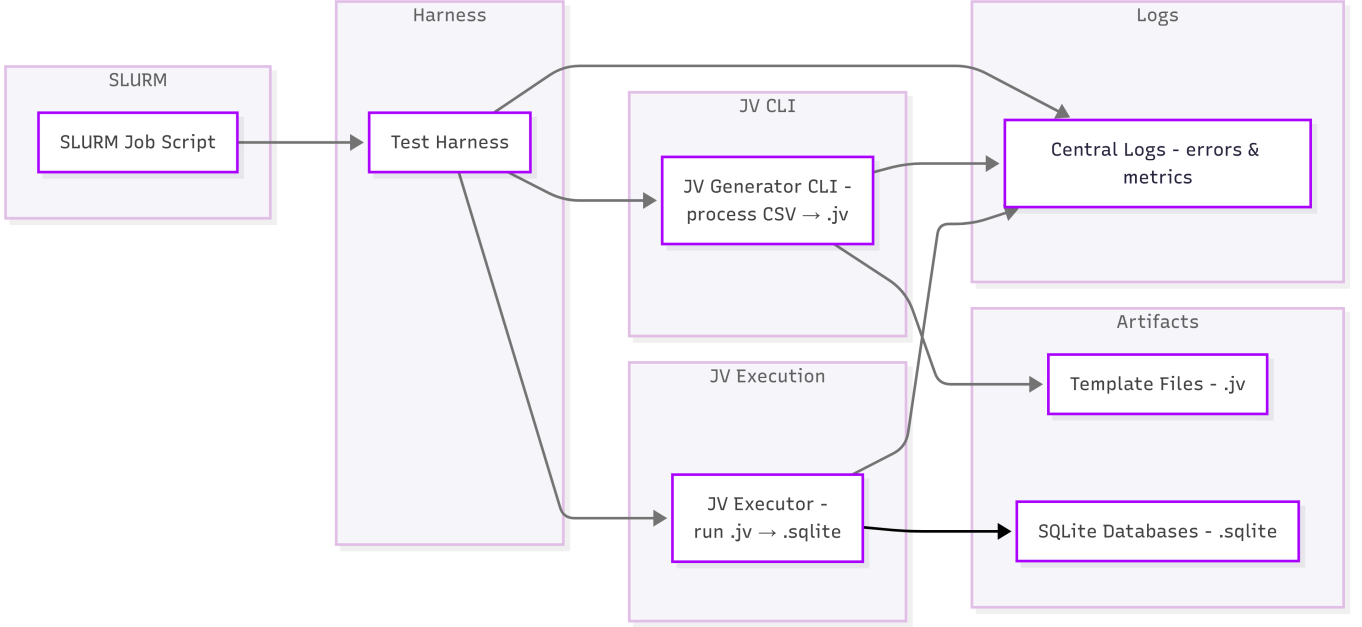


Figure 5: Test Architecture for Jayvee Template Generation

In this architecture, the SLURM job script first bootstraps the HPC environment—loading necessary modules, activating the Conda environment, exporting all required paths (including `CSVFiles/`, `JvFiles/`, `SQLiteFiles/`, `JsonFiles/`, and log file locations), and creating directories. It then invokes the Python test harness, which scans the input folder, calls the JV Generator CLI to infer schema and emit `.jv` templates, and immediately runs the JV Executor on each template to produce `.sqlite` sinks. Throughout this process, errors and execution metrics are aggregated into central log files for later inspection.

### 6.1.2 Edge Cases in CSV Schema Inference

When evaluating the Jayvee template generation system, we encountered several edge cases. 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) before applying Regular Expression matching. Columns with duplicate or missing headers should be resolved automatically by



assigning fallback identifiers (for example, `column_1`, `column_2`, and so on) to ensure that every field remains addressable. Special characters and multilingual content introduce further challenges: matching patterns in names like “résumé” or “order#id” requires Unicode-aware expressions and normalization to a standard form (such as NFKC) so that semantically equivalent strings compare correctly. In cases where a header suggests a numeric type—such as fields ending in `_id`—the system applies a tiered fallback strategy, defaulting to integer only if all values are strictly numeric, and otherwise inferring text to avoid errors from hybrid or malformed inputs.

Date fields often exhibit mixed formats even when the header indicates a date (for example, containing `_date`). Content-based validation must support flexible formats (including `YYYY-MM-DD`, `MM/DD/YYYY`, and written forms like `June 4 2021`) and either coerce values or flag rows that fail parsing. Numeric strings with leading zeros, such as ZIP codes or identifiers, must not be misinterpreted as numbers; when the column name suggests an identity field, the presence of leading zeros triggers a forced string type. Handling missing or special “null” values—such as `null`, `N/A`, or empty strings—requires semantic understanding: depending on context, these may be dropped, cast to `None`, or retained as literal strings.

Proper delimiter handling is also essential. Rather than simply splitting on commas, the system adheres to RFC 4180 (or a specified CSV dialect), especially for fields that include delimiters inside quoted text. Column name normalization—such as replacing hyphens with underscores—improves pattern-matching reliability for downstream consumers. Boolean inference must recognize a variety of representations (`1/0`, `true/false`, `YES/NO`) and normalize them into a single Boolean type, assuming no conflicting values exist.

All of these edge-case scenarios were incorporated into our evaluation process, verifying how both baseline heuristics and LLM-assisted inference generalize to structurally inconsistent or malformed CSV data. The results confirm that successful schema inference depends not only on accurate model predictions but also on robust preprocessing, normalization, and error-handling logic embedded throughout the parsing stack.

## 6.2 LLM Schema Inference Evaluation

### 6.2.1 Test Architecture Overview

The evaluation of Jayvee’s LLM-powered CSV header detection was grounded in a synthetically generated benchmark designed to emulate the inconsistencies, ambiguities, and pre-processing hurdles of real-world CSV data. Central to this process was the use of ChatGPT to generate a large and diverse corpus of 10,000 CSV files, each paired with an explicit ground truth annotation identifying the correct header row. These ground truth indices were embedded in JSON format and stored alongside their corresponding CSVs, enabling precise and automated comparison between model predictions and intended schema boundaries.

The CSV files were deliberately infused with heterogeneous pre-header content to simulate common real-world obfuscations. These included comments, legal disclaimers, metadata blocks, dates, units of measurement, multi-language headers, and other non-structural information that precedes the actual schema-defining row. This design choice introduced semantic ambiguity and structural irregularity into the dataset, challenging the models not simply to detect positional patterns but to exhibit genuine contextual understanding and schema discrimination. Each file thus acted as a stress test for both LLM-based parsing and prompt sensitivity.

The evaluation pipeline was executed in a high-performance compute (HPC) environment using SLURM to coordinate resource allocation and batch inference. The pipeline initializes by setting up the compute context: loading required modules, activating the Conda environment, and preparing directory structures for input CSVs, generated prompts, output predictions, logs, and results. Once initialized, a Python-based test harness iterates through all files in the LLMInf/CSVFiles/ directory, pairs each CSV with its associated JSON label from LLMInf/Labels/, and formats a prompt tailored to the model under test.

Inference is performed either via local engines (such as vLLM or HuggingFace Transformers) or through remote APIs compatible with OpenAI’s specification, depending on the model configuration. Once a prediction is received, the harness parses the result, extracts the proposed header row index, and compares it to the reference label. Accuracy statistics are continuously updated across the 10,000-file corpus, and all mismatches are logged for subsequent analysis, including token-level audit trails and full prompt histories to support reproducibility and debugging.

The following diagram in Figure 6 provides a structural overview of this evaluation loop:

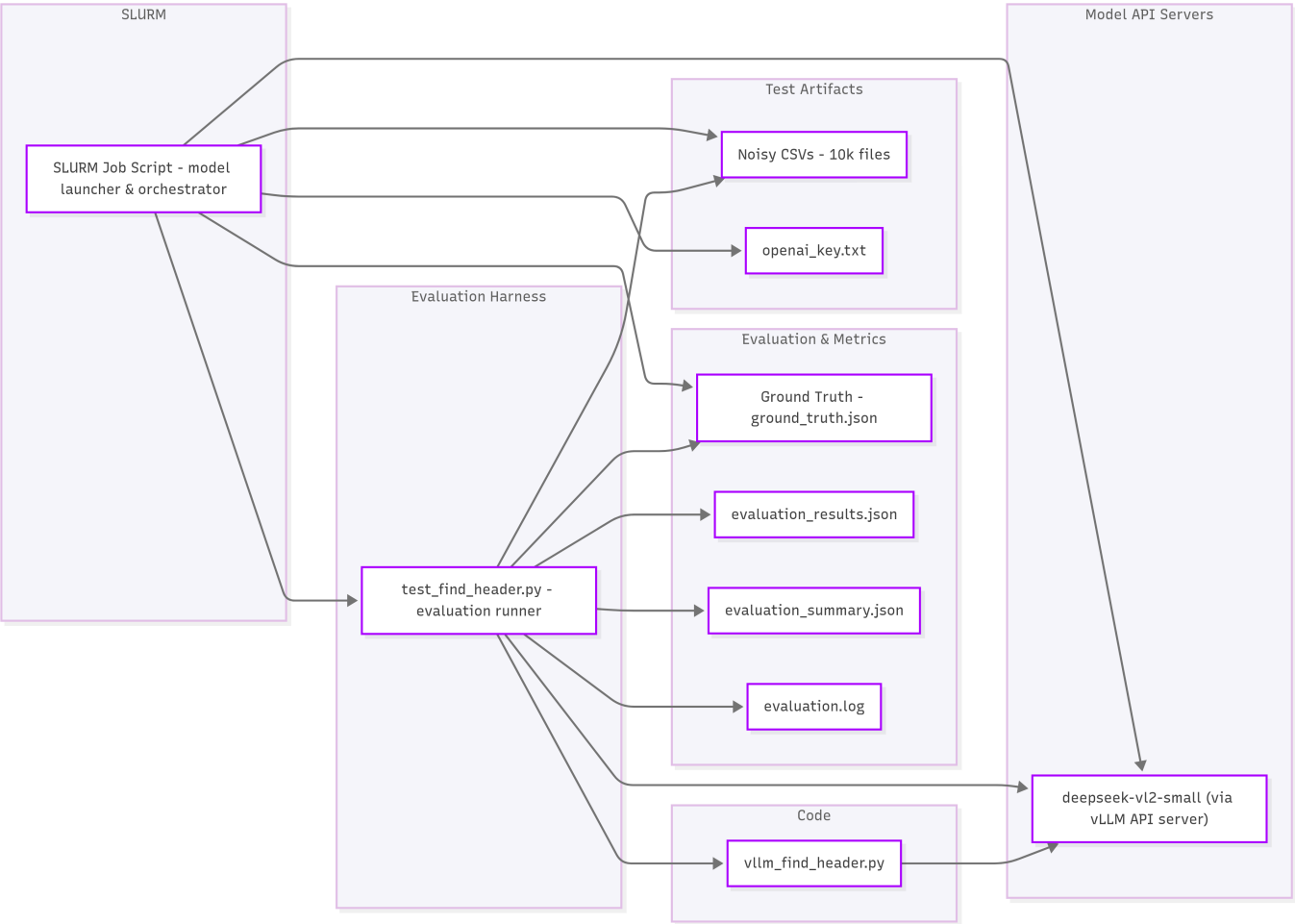


Figure 6: Test Architecture for LLM-based CSV Header Detection

The testbed’s design ensures that models are not only exposed to well-behaved files, but are rigorously evaluated under edge conditions that reflect practical deployment environments. The interplay of synthetic control (via ChatGPT), adversarial formatting (via pre-header noise), and automated ground truth comparison (via JSON indices) yields a robust and scalable platform for measuring the limits and strengths of LLM-based schema inference.

## 6.2.2 Evaluation Summary

Using the test architecture described above, multiple large language models were evaluated on their ability to correctly identify header rows within the 10,000 noisy CSV files. Accuracy was defined as the proportion of model predictions that exactly matched the annotated ground truth. Results varied widely across model families, with top-tier instruction-tuned models such as Deepseek-R1 achieving a high of 84.7% accuracy, correctly locating the header row in 8,470 of the test files. Less capable models—especially those without domain tuning—struggled to interpret pre-header noise and often defaulted to selecting the first or second row, resulting in accuracies as low as 62%.

A comparative assessment revealed that larger models consistently exhibited stronger contextual reasoning, especially when encountering extended metadata blocks or commentary preceding the actual schema. Their ability to recognize subtle lexical cues and structural intent allowed for more precise row selection. In contrast, smaller or instruction-naïve models frequently misinterpreted comment rows or misaligned multi-row headers, pointing to a lack of robustness in ambiguous or cluttered scenarios.

Error analysis surfaced several recurring failure modes. Files with inconsistent delimiters, embedded comments, or hybrid numeric-text columns often triggered incorrect inferences. Some models failed entirely to distinguish between a header and a descriptive unit row. In other cases, models overfitted to superficial patterns, misclassifying rows with frequent capital letters or delimiter density as the schema row despite lacking columnar semantics.

These findings suggest that while language models can significantly streamline schema discovery, especially when well-prompted and instruction-tuned, their reliability diminishes in the face of unstructured input. This underscores the importance of hybrid approaches that blend LLM predictions with traditional heuristics or rule-based prefilters. Additionally, fine-tuning on domain-specific CSV structures and systematic prompt optimization may offer further gains in detection fidelity.



## 7 Conclusions

### 7.1 Template Generation

I found that the Template Generation component reliably extracts and transforms CSV schemas into .jv templates, and my regex-based heuristics capture the essential structure of most input files. Implementing the tool as a desktop-only Tkinter application accelerated development and testing, but I acknowledge this choice limits the tool to GUI environments and prevents headless or web-based automation. While pandas-powered type inference handles batch processing smoothly, I did not optimize for low-latency or real-time performance in this initial version. Remote URL handling also proved a weak point: without robust sandboxing or validation, the system remains vulnerable. I opted for static, hardcoded block definitions to simplify early development, though this decision sacrificed runtime extensibility by omitting a plugin mechanism for custom block types. On the upside, the modular, function-level code structure has made maintenance and future enhancements straightforward. Comprehensive logging ensures I can trace and diagnose errors, and the deterministic, file-based I/O design has enabled reliable automated testing.

### 7.2 LLM Schema Inference

Through developing the LLM Schema Inference module, I explored a novel method for header-row detection using local large language models. This work revealed that the approach demands substantial computational resources and that even large, instruction-tuned models can hallucinate or misidentify headers in challenging cases. Although deploying inference on HPC hardware improved throughput, response times remain too

slow for interactive or real-time use. By building around powerful LLM backends, I achieved strong performance in controlled experiments, but this design limits adoption in resource-constrained environments. The system currently targets single-file workflows, and scaling to massive batch jobs will require additional orchestration and parallelization logic. From a security standpoint, sending raw CSV content directly to the model endpoint exposes potential data-leak and injection risks. Finally, the inherent non-determinism of LLM outputs has complicated reproducibility, highlighting the need for further work on prompt stabilization or ensemble strategies to produce consistent results.



## 8 Future Work

Several avenues remain open for enhancing both the template generation and LLM inference components. On the template side, migrating the core logic into a headless CLI or lightweight web service would enable seamless integration into automated ETL pipelines and cloud environments. Building a plugin framework for custom extractors, interpreters, and loaders would allow domain experts to extend block types without touching the core code, while performance profiling and parallelization of the pandas-based type inference and serialization steps could drive down latency and support larger datasets. Strengthening security around remote data ingestion—through sandboxed downloads, schema validation, and configurable network policies—would guard against malicious inputs and broaden adoption in enterprise contexts.

For the LLM schema inference module, improving accuracy and stability is paramount. Experimenting with ensemble prompts, retrieval-augmented generation, or adapter-based fine-tuning on a curated corpus of header-annotated CSVs could substantially reduce hallucinations and misclassifications. Developing lightweight, distilled model variants or on-the-fly model swapping would lower resource requirements and enable near real-time header detection. Scaling beyond single-file inference will require distributed orchestration—using frameworks like Dask or Ray to parallelize across multiple GPUs or nodes—and pairing heuristic validation rules with model predictions to catch and correct errors. Finally, expanding benchmark datasets to include multilingual and non-standard CSV formats will surface additional edge cases, guiding prompt refinements and model improvements that ensure robustness across diverse real-world scenarios.



# Bibliography

- [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>.
- [Gra25] I. Granite Team, “Granite-3.2-8B-Instruct: An 8-Billion-Parameter Long-Context AI Model with Controllable Thinking Capability,” *Granite Docs*, Feb. 2025, [Online]. Available: <https://huggingface.co/ibm-granite/granite-3.2-8b-instruct>
- [Dee25] DeepSeek-AI, “DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning.” [Online]. Available: <https://arxiv.org/abs/2501.12948>
- [Tea25] Q. Team, “Qwen3 Technical Report.” [Online]. Available: <https://arxiv.org/abs/2505.09388>
- [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. Nayeibi, 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>
- [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.
- [al.23] S. et al., “Aligning benchmark datasets for table structure recognition,” 2023.
- [Shi23] J. e. a. Shin, “Prompt Engineering or Fine Tuning: An Empirical Assessment of LLMs for Code (OpenReview),” 2023.
- [Wu+23] X. Wu, H. Chen, C. Bu, S. Ji, Z. Zhang, and V. S. Sheng, “HUSS: A Heuristic Method for Understanding the Semantic Structure of Spreadsheets,” *Data Intelligence*, vol. 5, no. 3, pp. 537–559, 2023, doi: 10.1162/dint\_a\_00201.
- [Fan+12] J. Fang, P. Mitra, Z. Tang, and C. L. Giles, “Table header detection and classification,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2012, pp. 599–605.
- [Mil18] N. Milošević, “A multi-layered approach to information extraction from tables in biomedical documents,” 2018.
- [Koc+16] E. Koci, M. Thiele, O. Romero, and W. Lehner, “A machine learning approach for layout inference in spreadsheets,” in *International Conference on Knowledge Discovery and Information Retrieval*, 2016, pp. 77–88.
- [BM20] S. S. Budhiraja and V. Mago, “A supervised learning approach for heading detection,” *Expert systems*, vol. 37, no. 4, p. e12520, 2020.
- [Bai+24] F. Bai, J. Kang, G. Stanovsky, D. Freitag, and A. Ritter, “Schema-Driven Information Extraction from Heterogeneous Tables,” *arXiv preprint arXiv:2305.14336v3*, 2024, [Online]. Available: <http://arxiv.org/pdf/2305.14336v3>
- [AI24] D. AI, “DeepSeek-R1: The First Fully Open Source RAG Language Model Suite,” *arXiv preprint arXiv:2501.12948*, 2024, [Online]. Available: <https://arxiv.org/abs/2501.12948>
- [Lij24] S. Lijin, “Every Way To Get Structured Output From LLMs.” [Online]. Available: <https://www.boundaryml.com/blog/structured-output-from-llms>
- [al.24] Z. et al., “Multimodal Table Understanding,” 2024.

- [Smo21] A. Smock Pesala, “PubTables-1M: Towards comprehensive table extraction from unstructured documents,” 2021.
- [Hea25] R. user Healthy-Nebula-3603, “Why is table extraction still not solved by modern multimodal models?.” 2025.
- [Ano24] Anonymous, “Knowledge-Aware Reasoning over Multimodal Semi-structured Tables,” 2024.
- [Des21] S. Desai Kayal, “TabLeX: A Benchmark Dataset for Structure and Content Information Extraction from Scientific Tables,” 2021.
- [UpS25] UpStage, “upstage/dp-bench dataset.” 2025.
- [Hel+25] P. Heltweg, G. Schwarz, D. Riehle, and F. Quast, “An Empirical Study on the Effects of Jayvee, a Domain-Specific Language for Data Engineering, on Understanding Data Pipeline Architectures,” *Software: Practice & Experience*, vol. 55, no. 6, pp. 1086–1105, 2025, doi: 10.1002/spe.3409.
- [YHC21] J. Yang, Y. He, and S. Chaudhuri, “Auto-Pipeline: Synthesizing Complex Data Pipelines By-Target Using Reinforcement Learning and Search,” *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2563–2575, 2021, doi: 10.14778/3476249.3476303.
- [KW19] S. Krishnan and E. Wu, “AlphaClean: Automatic Generation of Data Cleaning Pipelines,” 2019. [Online]. Available: <https://arxiv.org/abs/1904.11827>
- [Fen+17] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri, “Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 55–68. doi: 10.1145/3062341.3062351.
- [Ye+20] R. Ye, W. Shi, H. Zhou, Z. Wei, and L. Li, “Variational Template Machine for Data-to-Text Generation,” in *International Conference on Learning Representations (ICLR)*, 2020. [Online]. Available: <https://openreview.net/forum?id=HkejNgBtPB>



# Acronyms

**DSL** domain-specific language

**ETL** Ectract-Transform-Load

**GUI** Graphical User Interface

**LLM** Large Language Model





# Bill Of Materials

## Jayvee Template Generation

The Jayvee template generation pipeline, including the auxiliary schema inference module, relies on the following runtime environment and software dependencies:

- Python version: 3.8 or higher (validated with Python 3.10)
- Python packages:
  - `pandas==2.0.3`
  - `jsonpatch==1.33`
  - `python-dateutil==2.8.2`
  - `typing-extensions==4.6.0`
  - `pytz==2023.3`

The `tkinter` library is utilized for GUI-based folder selection dialogs and is part of the standard Python distribution (v3.8+). In addition, the pipeline depends on the `jv` command-line tool, which must be installed and available in the system's `PATH` for interpreting `.jv` files and generating intermediate SQLite databases.

## LLM-Based Schema Inference

A key enhancement to the pipeline involves automatic inference of JSON schema structures using a language model. This step leverages the `jsonformer` package to produce structured outputs conforming to a predefined schema.

This module requires:

- A transformer-based language model backend (e.g., HuggingFace-compatible model)

- Schema definitions in JSON Schema 2020-12 format
- Output post-processing with `jsonpatch` for iterative refinement

This component can be run independently or integrated into the larger generation pipeline. Its output is cached and reused to improve development efficiency and ensure stability of generated structures.

## Notes

- The listed Python packages are version-pinned to guarantee reproducibility and cross-platform compatibility.
- The development and testing environment was verified on *[insert OS here, e.g., Windows 10 / Ubuntu 22.04]*.
- The `python_requires` field in the `setup.cfg/setup.py` enforces minimum interpreter version requirements.