Jayvee Template Generation with LLM-Based Schema Inference

BACHELOR THESIS

Lea Buchner

Submitted on 1 of July 2025

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

Supervisor:
Johannes Jablonski
Prof. Dr. Dirk Riehle

FAU
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, 1 of July 2025

License

_____

Erlangen, 1 of July 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 development of a template generation system for Jayvee based on structured data inputs, more specifically, CSV files and (2) the evaluation of schema inference using locally hosted Large Language Models (**LLMs**).

The first 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. A large-scale evaluation of the template generator was conducted on 10,000 synthetic CSV files in an HPC environment using a custom SLURM-based test harness. Metrics include template validity, runtime, and SQLite output completeness.

The second 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-Coder, Qwen3-4B, and CodeLlama-7B-Instruct, all hosted local with Huggingface Transformers. 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, resulting in improved outcomes over naive prompting.

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

List of Figures

List of Tables

1 Introduction

As already mentioned in the abstract, this thesis explores schema inference through two complementary approaches: a rule-based Jayvee template generation system for structured data, and an evaluation framework for CSV schema inference using locally hosted **LLMs**.

Jayvee is designed to define, execute, and maintain data pipelines. It enables non-programmers to orchestrate complex data workflows using high-level constructs. However, a significant barrier to adoption lies in the manual effort required to create initial templates from raw data—especially when that data arrives in the form of messy or irregular CSV files.

The first part of this work introduces a system that automates Jayvee template generation from CSV inputs. It targets well-formed or mildly noisy files, using a deterministic parsing approach that skips malformed rows, extracts usable column headers, and emits ready-to-edit Jayvee programs. This system enables users with limited technical expertise to rapidly bootstrap pipelines from data they already possess.

The second part focuses on a more challenging problem: detecting the correct header row in CSV files with substantial noise, ambiguity, or structural irregularity. This task, central to reliable schema inference, is particularly difficult in real-world files that may contain metadata blocks, multiline comments, unit descriptors, or multilingual preambles. Traditional heuristic-based approaches—such as evenly spaced sampling, content snapshot selection, or rule-based parsing—tend to degrade significantly under these conditions (Jiang et al., 2021) [JVN21]. Recent work, however, has shown that LLMs prompted with structured instructions can outperform these heuristics in identifying headers under noisy conditions. Studies such as Sui et al. (2024)[Sui+24] and Liu et al. (2024) [Liu+24] report measurable gains in schema identification accuracy, with LLMs outperforming conventional baselines by 0.8% to 4.07% when carefully engineered prompts are used.

While the Jayvee template generator and the LLM-based evaluation system were developed and analyzed independently, their conceptual connection lies in their shared goal: automating the translation of imperfect, semi-structured data into structured pipeline-ready representations. Future work may integrate the two components, embedding LLM-based schema inference directly into the Jayvee generation flow to handle messier inputs end-to-end. For now, their modular separation allows for targeted improvements, easier experimentation, and clearer insights into each component's strengths and limitations.

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 intermediate JSON format serves as a modular and reproducible representation of the inferred pipeline logic. It decouples the data interpretation phase from the final Jayvee template generation, allowing for schema validation, unit testing, format conversion, and parallel execution. The JSON captures all pipeline semantics in a structured, portable way—ensuring that downstream processes like .jv rendering or debugging can operate independently and repeatably without re-parsing the original data.

This schema is subsequently converted into Jayvee format. A user-oriented Command-Line Interface (**CLI**) enables efficient file and URL selection, as well as batch processing of multiple CSV sources. Although no graphical interface currently exists, the CLI supports intuitive interaction patterns and structured input via command arguments or links files. Robust error handling and logging mechanisms ensure stable operation across diverse datasets and facilitate troubleshooting during execution.

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.

One of the evaluated models is DeepSeek-Coder, a code-centric transformer model optimized for program synthesis and structured data tasks. Although not explicitly instruction-tuned, its strong token-level precision and generalization on semi-structured inputs make it suitable for header detection in adversarial CSVs (Guo et al., 2024). [Guo+24]

The second model, Qwen3-4B, is a compact yet capable transformer from the Qwen3 family, trained with alignment techniques that improve instruction adherence and tabular comprehension (Yang et al., 2025). [Yan+25]

Lastly, CodeLlama-7B-Instruct, an instruction-tuned variant of Meta's CodeLlama model, was evaluated for its practical trade-off between scale, instruction fidelity, and local deployability. It is designed to follow natural language prompts and handle semi-structured data representations (Rozière et al., 2024). [Roz+23]

The following table summarizes the technical profile and empirical performance notes for each model in the context of this schema inference task:

Model	Parameters	Architecture	Key Capabilities
DeepSeek-Coder	Unknown (medium scale)	Code-focused transformer	High token-level precision, performs well on structured data (Guo et al., 2024) [Guo+24]
Qwen3-4B	4 billion	Instruction-tuned transformer	Lightweight yet instruction-aware, capable of handling structural noise (Yang et al., 2025) [Yan+25]
CodeLlama-7B-Instruct	7 billion	Instruction-tuned transformer	Designed for prompt following and semi-structured data understanding (Rozière et al., 2024) [Roz+23]

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 **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 **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 **LLMs**. It also bypasses the need for labeled datasets, leveraging the pre-trained model's knowledge directly without supervised training data (Pornprasit et al., 2024; Shin et al., 2025). [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 (Zhang et al., 2024) [Zha+24]. In code-domain benchmarks, careful prompting strategies have demonstrated competitive BLEU scores compared to fine-tuned models, especially when human feedback is integrated conversationally (Smock et al., 2023; Shin et al., 2023) [Shi+23, SPA23]

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 (Zhang et al., 2024). [Zha+24]

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 et al., 2023) [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 (Fang et al., 2012) [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 (Milošević, 2018) [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 et al., 2023) [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 (Milošević, 2018) [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 (Koci et al., 2016). [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 (Budhiraja & Mago, 2020) [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.

While machine learning strategies like Strudel and the work by Khusro et al. enhanced schema inference by combining supervised classification with heuristic post-processing — they still relied on labeled training data and handcrafted rules — modern **LLMs** offer a transformative advancement.

2.2.3 Modern Techniques Leveraging Language Models

Recent advancements have brought **LLMs** into schema inference workflows, enabling powerful new capabilities in table understanding. In Schema-Driven Information Extraction from Heterogeneous Tables, Bai et al. (2024) reformulate schema inference as a structured extraction task: given a JSON schema and a noisy table, instruction-tuned **LLMs** like GPT-4 and Code-Davinci output structured JSON records. They report impressive F1 scores from 74.2% to 96.1%, matching or exceeding traditional supervised systems—all without any task-specific training (Bai et al., 2024). [Bai+24]

Further, Wu et al. (2025) apply LLMs to taxonomy inference over table data, showing these models consistently reconstruct hierarchical entity types that align closely with gold-standard ontologies, outperforming classic lexical-structural methods in consistency and semantic accuracy (Wu et al., 2025). [WCP25]

These instruction-following transformers thus provide schema-aware parsing across contexts and noise levels—even handling multi-row headers or missing delimiters—far surpassing feature-engineered ML pipelines.

2.2.3.1 Retrieval-Augmented Language Models and Context Injection

Retrieval-augmented language modelss (**RALMs**) like DeepSeek Coder enhance reasoning and disambiguation in data tasks by integrating external knowledge sources at inference time. DeepSeek Coder 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. This augmentation strategy allows models to dynamically pull in domain-relevant information, which improves both factual accuracy and interpretability in schema inference tasks (Deepseek-AI, 2024). [Dee24]

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. This makes retrieval-enhanced models significantly more robust than static models that rely solely on training-set priors. Practical implementations, including experiments involving DeepSeek Coder with LangChain or LangGraph frameworks, have demonstrated that recursive retrieval pipelines improve disambiguation and classification in real-world CSV tasks (Deepseek-AI, 2024). [Dee24]

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 Coder use retrieval-augmented generation (**RAG**) to externalize parts of the reasoning process, leading to better generalization in unseen formats or edge cases. This makes them especially suited to schema inference applications,

where file variability, semantic ambiguity, and format drift are common (Deepseek-AI, 2024). [Dee24]

2.2.3.2 Multi-modal Models for Tabular Data Analysis

Recent work has focused on large multimodal models such as Qwen3-4B, which can jointly process text, table structures, and images. These models demonstrate strong capabilities in understanding complex table layouts and semantic content by integrating visual and textual modalities. 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 requiring specialized architectures (QwenLM, 2024). [Qwe24]

Nonetheless, even state-of-the-art multimodal LLMs like Qwen3-4B struggle with reconstructing table structures reliably, especially from images alone, highlighting the necessity for improved visual grounding and spatial reasoning. [Qwe24]

2.2.3.3 Instruction-Tuned Models for Code and Table Tasks

Instruction-tuned language models, such as CodeLlama-7B-Instruct, demonstrate strong performance in both code understanding and schema-oriented prompting. CodeLlama-7B-Instruct is a variant of Meta's Code Llama family, fine-tuned to follow natural language instructions rather than only generate code. It inherits from the Llama 2 architecture and was further trained on approximately 500 billion tokens of code, followed

by 5 billion tokens of instruction tuning to improve compliance with developer directives (Rozière et al., 2024; Meta AI, 2024). [Met24, Roz+23]

As an autoregressive transformer, CodeLlama 7B-Instruct supports code completion, infilling between code snippets, and instruction-driven generation across multiple languages—including Python, C++, JavaScript, and more. Critically, it features a long context window of up to 100k tokens, enabling it to handle large tables or long scripts in a single prompt—an essential capability for batch schema inference tasks (Meta AI, 2024). [Met24]

Meta-released benchmark evaluations on HumanEval and MultiPL-E datasets indicate that CodeLlama models achieve state-of-the-art performance among open-source models, with pass@k scores comparable to LLaMA alternatives. The instruction-tuned variant further improves functional accuracy and compliance, making it suitable for structured generation—such as converting CSV headers into DSL schema definitions (Rozière et al., 2024). [Roz+23]

Importantly, CodeLlama-7B-Instruct supports fill-in-the-middle infilling and boasts a long-context window of up to 100k tokens, enabling it to generate, explain, and manipulate large code blocks or tables in one prompt—an essential capability for schema inference workflows (Meta AI, 2024). [Met24]

Performance benchmarks show CodeLlama-7B-Instruct achieves near-state-of-the-art results among open-source code models. The full CodeLlama family outperforms comparable LLaMA-2 baselines on benchmarks such as HumanEval, MBPP, and MultiPL-E, with the "Instruct" variant demonstrating enhanced functional accuracy and instruction compliance—qualities ideal for structured schema generation in tabular and DSL contexts. [Met24]

In schema inference, this model's instruction-tuned design allows prompts like "extract header row as JSON schema" or "fill missing schema cell based on context," enabling interpreters to process complete table segments in a single pass. Its long-context capability avoids errors from prompt truncations. [Met24]

2.2.4 Extending Prior Work: Local LLMs for Noisy CSV Header Detection

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

While traditional heuristic and rule-based systems offer some utility in idealized contexts, their effectiveness sharply declines under the noise and irregularity characteristic of real-world CSV files. Instruction-following **LLMs** such as DeepSeek Coder, Qwen3-4B, and lCodeLlama-7B-Instruct demonstrate promising zero-shot capabilities in discerning header rows and tabular semantics, even under structural ambiguity [Dee24, Met24, Qwe24]. Locally hosted **LLMs** offer privacy-preserving and cost-effective schema inference workflows that operate entirely offline, an underexplored but increasingly important deployment paradigm [Dee24].

Moreover, while much current research emphasizes cloud-based LLM systems, this work evaluates model performance in a local execution context, where hardware limitations and model optimization constraints present unique challenges. This brings empirical clarity to the practical feasibility of deploying such models for tabular understanding outside of centralized, compute-rich infrastructures [Dee24, Met24, Qwe24].

The thesis also introduces a template generation engine for Jayvee pipelines, extending focus from structural understanding to operational synthesis. By treating noisy tabular inputs as both targets for structural interpretation and sources for generating executable, user-friendly transformation pipelines, it proposes a holistic approach to data usability engineering for users with limited technical expertise.

Together, this work advances the field by (1) evaluating LLM generalization boundaries in schema inference under realistic constraints, and (2) designing a DSL-centric automation layer that translates understanding into actionable pipelines, offering a practical system architecture for next-generation tabular data interfaces.

2.3 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 (Heltweg et al., 2025) [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 (Yang et al., 2021) [YHC21].

Similarly, AlphaClean explores generative frameworks that combine operator libraries with quality metric-driven search, producing robust cleaning pipelines without manual scripting (Krishnan et al., 2019) [KW19].

Earlier work on component-based synthesis in PLDI-style systems— so named for their roots in the Programming Language Design and Implementation (**PLDI**) research community—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 (Feng et al., 2017) [Fen+17]. These systems typically emphasize modularity, compositionality, and example-driven synthesis, leveraging structured DSLs and formal semantics to generate programs that satisfy user-specified constraints or data transformations. Jayvee, by using templates and schema-driven synthesis, aligns with this ethos—especially in how data schemas map to composable pipeline fragments, enabling non-expert users to build robust workflows programmatically.

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 et al., 2020) [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.

2.4 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 **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 to make it parsable by downstream components. To enhance model performance under zero-shot conditions, the approach incorporates prompt engineering techniques tailored to guide the model output.

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 result quality and reliability. Evaluation is grounded on JSON-labeled synthetic benchmarks with controlled noise and ambiguity.

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

In this chapter, the architecture of both the Jayvee Template Generation and the LLM Schema Inference subsystems are presented, illustrating how each layer and component interacts to fulfill its respective data-processing goals.

4.1 Jayvee Template Generation

The Jayvee Template Generation system automates the creation of structured Jayvee pipeline templates by extracting metadata—such as column names, types and overall schema—from heterogeneous CSV datasets. Although implemented as a single **CLI**-driven script, its internal architecture follows a modular, layered component model that mirrors data-flow principles common in modern ETL tooling. At the top sits the orchestration layer, embodied by a command-line interface that parses user arguments to determine whether inputs come from a single CSV file, a URL, a list of URLs, or an existing JSON schema; it also configures output directories and flags for optional JSON schema emission. Once arguments have been interpreted, the execution controller routes control flow to the input layer.

The input layer encompasses a CSV/URL reader for fetching raw data, a links-file processor for iterating over multiple URLs in a text file, and a file-handling utility responsible for normalizing paths, sanitizing names and validating resource availability. After raw data has been ingested, the template generation engine takes over. In its ingestion and preprocessing stage, raw CSV content is parsed into a DataFrame with correct quoting and whitespace handling; unnamed columns are automatically renamed according to simple heuristics that distinguish strings from numerics. The cleaned data then flows into the type inference component, which leverages Pandas dtypes and custom mappings to derive Jayvee-compatible types such as boolean, integer, decimal or text.

These inferred types feed into the schema construction stage, which assembles a JSON schema conforming to Draft-07: it defines pipeline blocks (extractors, interpreters, loaders and optional header-writers) and pipes (the directed edges that represent data flow).

Once the JSON schema is complete, the Jayvee conversion component translates it into a Jayvee file by emitting Jayvee DSL code with explicit pipeline, block and arrow-based flow syntax. If the CREATE_JSON flag is enabled, the generated JSON schema is also saved as an intermediate artifact. Throughout every stage—from ingestion through conversion—a centralized error logger captures exceptions and writes them to a persistent error_log.txt, enabling post-mortem debugging without halting batch processing.

The result is a pair of generated artifacts: a production-ready Jayvee pipeline template suitable for execution by the Jayvee interpreter, and, when requested, a machine-readable JSON schema. This layered, component-based architecture ensures clear separation of concerns, straightforward extensibility (for example, adding new interpreter types or validation stages), and robust error handling, all while maintaining an internal structure that aligns with enterprise ETL frameworks.

The following Figure 1 shows the component diagram that visualizes this architecture:

The architecture is depicted in the following diagram:
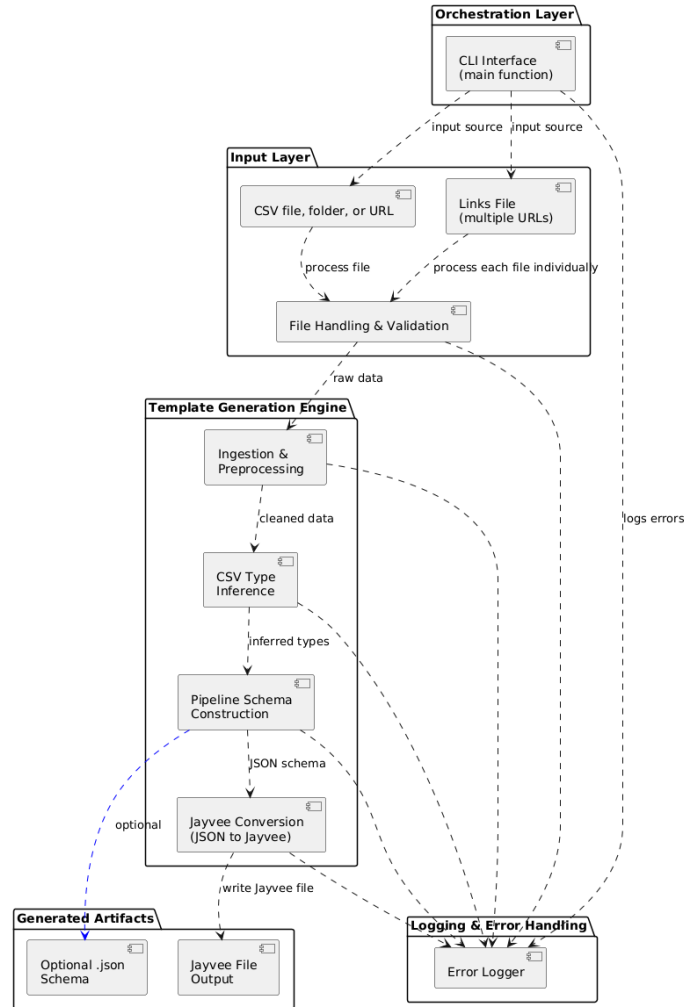
Figure 1: Architecture Component Diagram - Jayvee Template Generation

The diagram illustrates these layers, their responsibilities and the flow of data and control between them. The orchestration layer sits at the top, directing input sources toward the input layer; the input layer prepares data for the core engine, which itself is subdivided into ingestion, inference, schema construction and conversion stages; finally, generated artifacts emerge at the bottom while a cross-cutting logging component captures errors at every step. This visualizes how each component contributes to the end-to-end transformation from raw CSV inputs to executable Jayvee pipelines, how optional JSON schema emission is handled, and how robust error handling is woven throughout the process.

4.2 LLM Schema Inference

The LLM Schema Inference system is built as a multi-layered, Slurm-orchestrated pipeline that transforms raw tabular datasets into structured metadata by leveraging large language models. At its highest level, a Slurm job script serves as the orchestration layer: it submits an array of tasks across GPU nodes, configures environment modules (Python, CUDA), activates a dedicated Conda environment, and ensures each Slurm array task isolates a specific model and data shard. Once a job is dispatched, the execution controller invokes evaluate.py, which parses command-line arguments (model directory, stride, offset, parallelism) and initializes distributed process groups when GPUs are available.

In the input layer, evaluate.py locates and validates CSV files from a designated directory, loads ground-truth annotations from JSON, and shards the file list across ranks and strides for data parallelism. Each selected CSV path, paired with its model identifier and an environment mapping of CUDA device assignments, is passed to a multiprocessing pool that executes run_find_header_job tasks in parallel.

Within the inference layer, each worker invokes a subprocess running find_header.py, which itself loads the first 20 lines of a CSV as text, constructs a prompt enriched with examples, and calls a local Hugging Face transformer model (via AutoTokenizer and AutoModelForCausalLM) to predict the header row. The model response is logged, cleaned of extraneous text, parsed into JSON using a Pydantic schema, and emitted back to the parent process.

The evaluation layer consumes these JSON-encoded predictions alongside the ground truth. It tallies parsed files, missing annotations, correct and incorrect inferences, computes an accuracy percentage, and aggregates detailed records into a per-model summary. Partial results—both summary metrics and individual file evaluations—are serialized to disk under partial_results. A final aggregation step then merges these per-model JSON outputs into a consolidated report, which can be consumed downstream or visualized separately.

Throughout every stage—from Slurm orchestration through model inference and evaluation—a rotating file logger captures informational messages, warnings, and errors without interrupting the pipeline, thereby enabling fault isolation and continuous batch processing.

Below in Figure 2 this layered architecture is illustrated by a component diagram icluding the flow of control and data between components as well as the cross-cutting logging concern:
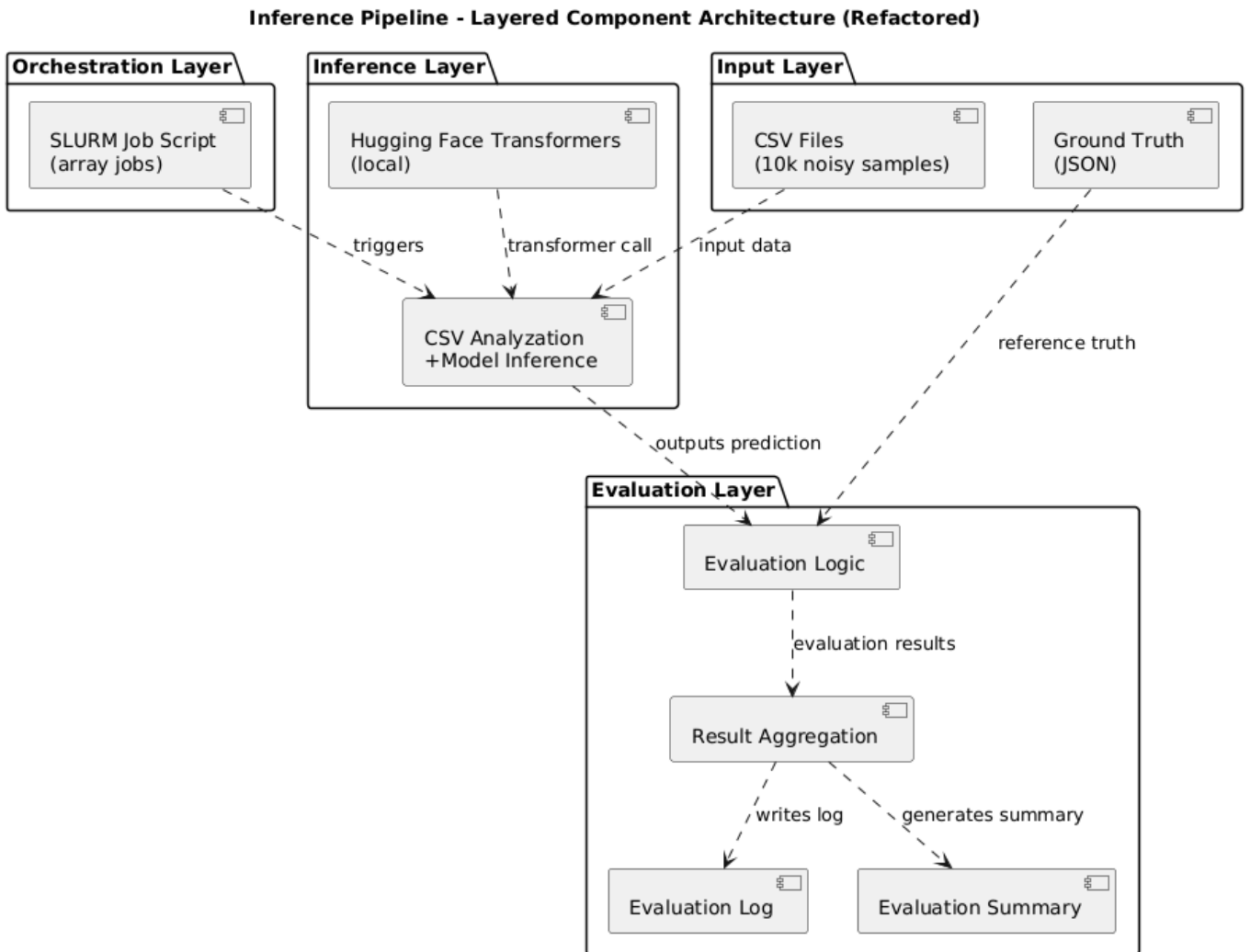
Figure 2: Architecture Component Diagram – LLM Schema Inference

By depicting the Slurm script at the top, the diagram emphasizes that each array job fully encapsulates its own model instance and GPU allocation. The input layer's separation of raw CSV loading and ground-truth ingestion clarifies data validation duties, while the inference layer splits control logic from per-file modeling. The evaluation layer then systematically computes and aggregates results, and a dedicated logging component captures runtime behavior. This modular, layered design ensures that new models, alternative parsing strategies or enhanced evaluation metrics can be introduced independently, while Slurm's array mechanism guarantees horizontal scalability and fault isolation across high-performance computing environments.

5 Implementation

This section describes how each subsystem is realized in code, mapping design to concrete Python scripts and workflows.

5.1 Jayvee Template Generation

The Jayvee Template Generation Script transforms CSV inputs into Jayvee pipeline templates via a layered ETL process, all driven from the command line.

5.1.1 Design Overview

The Jayvee 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 Jayvee format, enabling downstream processing in structured data integration workflows, such as automated database population, analytical pipeline configuration, or data quality validation.

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.

The system begins with the user initiating an operation via **CLI**, 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 and saved in Jayvee. 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 **CLI** Component presents the user interface, handling interactions such as file selection, URL input, and links-file browsing. A links file refers to a plain-text file containing a list of remote data sources (e.g., CSV URLs), allowing users to batch-load and preview multiple resources through the interface. The **CLI** enables users to browse these entries interactively and choose specific links for processing. 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 to Jayvee datatypes: 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 & Jayvee Writer Component saves the pipeline structure as a JSON file (when enabled), and always writes the corresponding Jayvee 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. CamelCase naming for pipeline and block identifiers is derived from filenames or URLs using a utility function. Spreadsheet-style column labels (A1, B1, …) are generated for unnamed headers via a separate index-mapping utility.

5.1.3 Edge Cases in CSV Schema Inference

During the implementation of the Jayvee template generation system, I focused on several key edge cases to ensure robustness in CSV template generation. Columns with missing or default "Unnamed" headers are automatically renamed based on the first data value in each column—for example, renamed to "col_name" if the first value is a string, or "col_value" if numeric—so that every column remains addressable. Leading and trailing whitespaces around column headers are removed to prevent mismatches during processing. The CSV parser respects quoted fields containing delimiters by using appropriate options such as a quote character and skipping initial spaces, adhering to common CSV dialect conventions rather than simply splitting on commas. Column data types are inferred from pandas' dtype detection and mapped Jayvee datatypes including boolean, integer, decimal, and text; notably, datetime types are coerced to text to handle their variability. When column headers are renamed, the pipeline schema dynamically includes an intermediate step to write these corrected headers, ensuring downstream components consume consistent column names. Boolean types are inferred from pandas native detection, although explicit normalization of different boolean representations was not implemented beyond this. Additionally, the system normalizes file names extracted from file paths or URLs into CamelCase for consistent naming of pipelines and processing blocks. Throughout processing, errors during CSV reading or JSON file saving are logged

with timestamps to facilitate troubleshooting without interrupting the overall workflow. While more complex edge cases such as Unicode normalization for multilingual headers, flexible date format coercion, and semantic handling of null values were identified as challenges, they remain unimplemented in the current system iteration.

5.1.4 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 transformed into a JSON pipeline description. When the CREATE_JSON environment flag is enabled, this JSON is persisted to disk alongside the default Jayvee output. The final Jayvee output format is a human-readable, line-based representation of the extraction template.

5.1.5 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 Jayvee output format is a human-readable, line-based representation of the extraction template.

5.1.6 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. All critical errors, including malformed inputs or I/O failures, are logged to a centralized `error_log.txt`. Structured error reporting avoids

stack trace noise and supports batch evaluation. **(?? GUI ??)** alerts present user-friendly messages while the **CLI** remains quiet unless run interactively.

5.1.7 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 Jayvee 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 Jayvee 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:
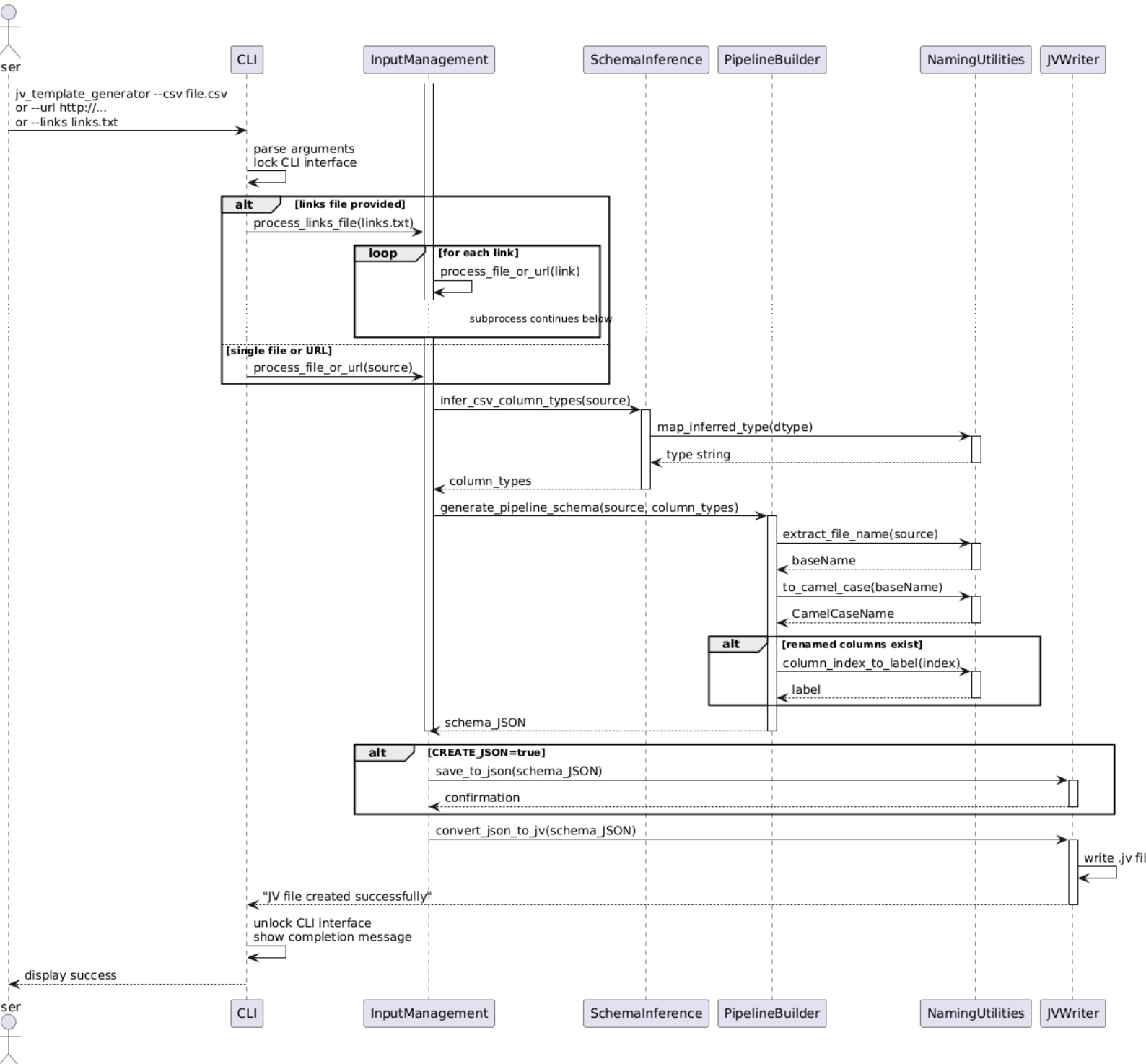
Figure 3: Sequence Diagram - Jayvee Template Generation

The sequence diagram above delineates the end-to-end implementation workflow of the Jayvee Template Generation Script, tracing each interaction from the moment a user initiates the process through to the successful creation of a Jayvee 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 & Jayvee Writer first persists the intermediate schema to disk; regardless, it then converts the JSON representation into the final Jayvee 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.

An auxiliary evaluation script, test_pipeline_generation.py, supports batch testing and performance profiling. It ingests CSVs from a directory, invokes jv_template_generation.py as a subprocess, verifies generation of Jayvee templates, and executes them via the Jayvee **CLI** to produce SQLite outputs. Metrics such as execution time, return codes, and output file status are logged centrally. The script supports parameterization via **CLI** flags for parallelism (–parallel) and sub-sampling (–every-nth).

5.2 LLM Schema Inference

This section details the offline, Slurm-driven pipeline for using LLMs to infer CSV header rows at scale.

5.2.1 Design Overview

The LLM-based schema inference subsystem addresses the task of identifying the header row in noisy or ambiguous CSV files using an autoregressive transformer model executed entirely offline. The system is designed for scale, repeatability, and modularity, enabling benchmarking across multiple models on large input corpora without internet access or server dependencies.

At its core, the system processes each CSV file using a standalone script (find_header.py) that prepares a structured prompt using few-shot examples, executes inference via a Hugging Face model, and returns a machine-readable JSON output. The workflow is orchestrated through Slurm job arrays (parallel_evaluation.sh), where each task handles one model independently. Evaluation results are serialized to disk, allowing partial recovery, reproducibility, and aggregate analysis.

This architecture replaces earlier API-driven designs with fully **CLI**-based workflows that are portable and privacy-preserving. Inference is conducted directly within the Python runtime using locally stored model weights. All outputs conform to a strict schema for downstream compatibility, and each stage is instrumented with structured logging to support debugging, validation, and performance tracking.

5.2.2 Component Responsibilities

The evaluation process begins by reading the first 20 lines of each CSV file. This sampling strategy captures enough structural variation and noise to provide meaningful context while keeping the prompt size within model token limits. Once read, the snippet is passed to find_header.py, which appends it to a few-shot prompt template consisting of three curated examples. The prompt is crafted to follow a consistent and readable layout, with newline padding and clearly delimited JSON answers.

Inference is executed locally using the Hugging Face transformers library. The model and tokenizer are loaded directly from disk using standard APIs such as AutoModelForCausalLM and AutoTokenizer. To enforce determinism, the decoding parameters are fixed: the temperature is set to zero, maximum token limits are specified, and if applicable, a stop sequence is used to terminate generation cleanly. These settings ensure consistent outputs regardless of the underlying hardware or parallelism configuration.

After generation, the model's response is expected to contain a valid JSON object with two keys: columnNameRow, indicating the 1-based index of the header row, and Explanation, providing natural language reasoning for the choice. The response is parsed using Python's standard json library. If the model output is malformed or extraneous text appears before the JSON block, the parser attempts to extract the first valid JSON-like substring using regular expressions. Any parsing failures trigger a fallback mechanism that returns a default result and logs the incident for later analysis.

Evaluation is carried out per model using evaluate.py, which applies this inference process across a large CSV corpus and records whether each prediction matches the corresponding value in a ground_truth.json file. All predictions, explanations, correctness flags, and metadata are written to disk in structured form. These intermediate results are aggregated post-hoc using aggregate_results.py, which merges per-model results into global summaries and evaluation metrics.

5.2.3 Data Flow

The evaluation pipeline initiates with directory traversal to locate all target CSV files, typically capped at 10,000 instances for large-scale benchmarking. For each file, a 20-line preview is extracted and sent to find_header.py, where a prompt is generated by inserting this snippet into a few-shot template. The transformer model processes the prompt and produces a structured response.

This response is then parsed and compared to a reference value from the ground truth dataset. The outcome is logged, with both the predicted and expected values recorded. If parsing fails or the result is invalid, the system logs the error, substitutes a default row index (typically 1), and attaches a diagnostic message. This process continues for each file in the corpus.

After all files have been evaluated for a given model, the results are saved in a partial results directory. When all models have been processed—typically via Slurm job arrays—aggregate_results.py is executed to compile the individual logs into a comprehensive results file (evaluation_results.json) and a summary statistics file (evaluation_summary.json).

5.2.4 I/O Formats & Schemas

The input to the system consists of a plain text string made up of the first 20 lines of a given CSV file. This preview may include noise such as comments, blank lines, or inconsistent delimiters, and is not sanitized prior to processing. This design choice reflects the real-world messiness of CSVs and challenges the model to generalize beyond cleanly formatted input.

The output expected from the model is a strict JSON object containing two fields. The columnNameRow field holds an integer indicating the row where the column headers appear, using 1-based indexing. The Explanation field is a free-form natural language string that justifies the model's choice. This output is parsed with the standard json

module and validated against these schema constraints. If any structural or semantic issues are detected, the system returns a fallback output along with a log entry that includes the raw response for debugging purposes.

This consistent format facilitates integration with possible downstream workflows, such as automated template generation, performance analytics, or error inspection tools. The structure also ensures compatibility with a wide range of evaluation and post-processing utilities.

5.2.5 Technology Stack

The implementation is written in Python 3.12 and uses Hugging Face's transformers library to load and run language models offline. Inference is conducted entirely within the Python runtime, eliminating the need for any API servers or internet access. The evaluate.py and aggregate_results.py scripts are modular and self-contained, designed for efficient batch execution on high-performance clusters. Logging is handled through Python's built-in logging module, which outputs both human-readable and machine-readable logs for debugging and analysis.

The system architecture is fully portable and compatible with a variety of computing environments, ranging from local workstations to Slurm-managed GPU clusters. Models are loaded from disk without requiring Hugging Face Hub access, and no tokenizers or resources are fetched remotely during runtime. This guarantees operational stability in air-gapped or privacy-sensitive environments.

5.2.6 Error Handling

The system is designed with layered exception handling at every stage. If a CSV file is unreadable or improperly formatted, the corresponding error is logged and the file is skipped. When the model returns an output that fails to parse as JSON, a regular expression is used to extract the most likely JSON block. If parsing still fails or required

fields are missing, a default result is generated and recorded. These fallbacks include the assumed header row (typically the first row) along with an explanation of the error, allowing users to trace and debug failures systematically.

All logs are serialized to disk and grouped by model, making it easy to review the performance of individual models and track recurring issues. The design ensures that evaluation is never halted due to a single failure. Instead, the system continues processing remaining files while isolating and documenting any errors encountered along the way. This robustness is critical for large-scale model benchmarking and reproducible experimentation.

5.2.7 Implementation Details

The overall evaluation process is illustrated by the sequence diagram below. It captures the orchestration between Slurm, the individual scripts, and the modular components of the schema inference workflow.
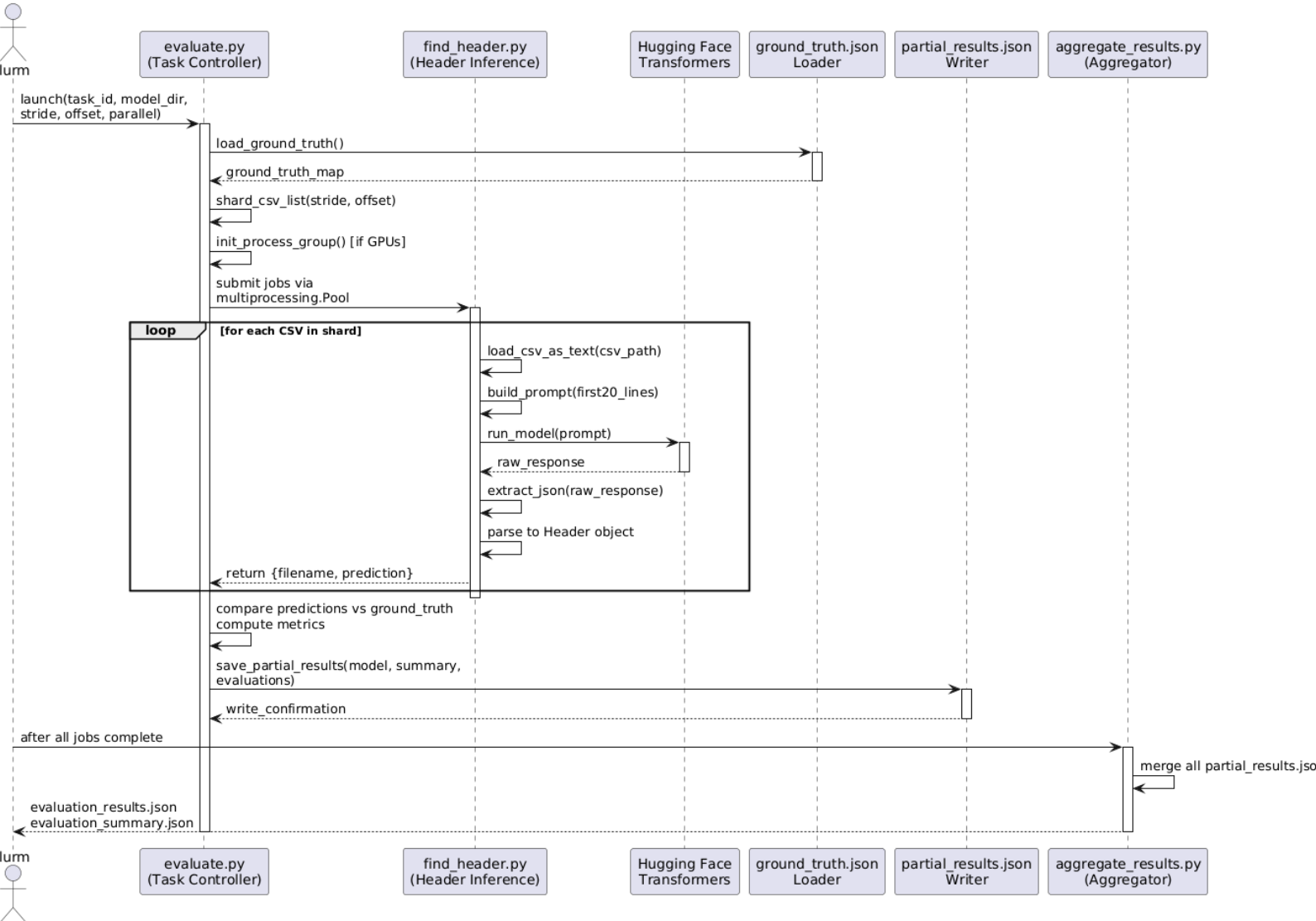
Figure 4: Sequence Diagram – LLM Schema Inference Evaluation Pipeline

The evaluation begins with the submission of a Slurm job array using the parallel_evaluation.sh script. Each array task selects one transformer model from a predefined list and runs the evaluate.py script in an isolated environment. This script reads all target CSV files, invokes find_header.py for each one via a subprocess, and compares the model's prediction to the known ground truth. Results are written to model-specific JSON files inside the partial_results directory.

Once evaluation is complete for all models, the aggregate_results.py script is executed. It reads each partial results file, merges the per-file evaluations, and produces two output files: evaluation_results.json, which contains the full set of predictions, and evaluation_summary.json, which summarizes accuracy and statistics for each model.

This architecture supports graceful degradation, parallel execution, modular restarts, and detailed logging. It is optimized for batch operation and reproducible benchmarking, making it well-suited for systematic evaluation of header inference capabilities in large language models.

6 Evaluation

This chapter presents the testing frameworks and results for both the Jayvee template generation and the **LLM** schema inference pipeline, demonstrating system behavior at scale and under edge-case-specific constraints.

6.1 JV Template Generation Evaluation

This section describes the high-performance test harness used to validate and benchmark the Jayvee template generation script across 10,000 structurally consistent CSV files, as well as a separate suite of edge-case scenarios designed to test robustness.

6.1.1 Test Architecture Overview

To ensure both correctness and scalability, an automated testing infrastructure was deployed on an **(?? HPC ??)** system. The 10,000 CSV files under `CSVFiles/` were processed one by one through the Jayvee generator to emit `.jv` templates, which were then immediately executed to produce corresponding `.sqlite` sinks.

```
----------------------------------------

— Final Summary —
Total CSV processed         : 10000
Total generation time       : 9117.525 seconds
Average generation time     : 0.912 seconds
JV files generated (this run) : 10000
Total JV files existing      : 1000
Total execution time         : 78363.090 seconds
Average execution time       : 78.363 seconds
JV executions succeeded      : 1000
JV executions failed (rc≠0)  : 0
Empty SQLite files           : 0
Execution success rate       : 100.00%
Overall JV→SQLite yield      : 10.00%
========================================
iwos101h@tinyx:~/TemplateGeneration$ ▓
```

Figure 5: Test Architecture for Jayvee Template Generation

A SLURM job script bootstrapped the environment by loading necessary modules, activating the Conda environment, exporting required paths (`CSVFiles/`, `JvFiles/`, `SQLiteFiles/`, etc.), and creating output directories. The Python test harness then scanned the input folder, invoked the Jayvee CLI for schema inference, and ran the resulting `.jv` files via the Jayvee Executor. All logs (stdout, stderr) and result artifacts were centrally aggregated for analysis.

6.1.2 Evaluation Results and Execution Validation

The evaluation of the template generator is centered not just on template creation, but on the ability to execute each generated pipeline successfully. This integrated validation ensures the templates are not only syntactically correct but also functionally operational.

All 10,000 `.jv` templates were executed end-to-end, each invoking a full Jayvee pipeline: extraction, interpretation, and final loading into a SQLite sink. Execution logs were captured, and the presence of valid SQLite outputs was programmatically verified.

```
------------------------------------------
—— Final Summary ——
Total CSV processed           : 10000
Total generation time         : 9117.525 seconds
Average generation time       : 0.912 seconds
JV files generated (this run) : 10000
Total JV files existing       : 1000
Total execution time          : 78363.090 seconds
Average execution time        : 78.363 seconds
JV executions succeeded       : 1000
JV executions failed (rc≠0)   : 0
Empty SQLite files            : 0
Execution success rate        : 100.00%
Overall JV→SQLite yield       : 10.00%
==========================================
iwos101h@tinyx:~/TemplateGeneration$ ▌
```

Figure 6: Evaluation Summary for Jayvee Template Generation

The results:

- 100% execution success rate across 10,000 pipelines
- No empty SQLite outputs or runtime errors encountered
- Average generation time per file: under 1 second
- Total generation time for 10,000 templates: 2.5 hours

These results confirm the structural soundness of the Jayvee schema inference process. Although the full corpus did not contain highly noisy or malformed CSVs, the test still covered a broad range of consistent, real-world formats. The successful execution of each

pipeline validates that the emitted templates are not only syntactically correct but semantically operable, highlighting the system's readiness for scaled production deployment.

Execution success functioned as both a benchmark and a quality filter: any misalignment—such as broken headers, invalid types, or parsing failures—would have led to execution errors or incomplete database sinks. The absence of such issues affirms the generator's ability to synthesize valid, well-formed pipelines from structurally sound inputs.

6.1.3 Edge Cases in CSV Schema Inference

To further stress-test the pipeline's robustness, a dedicated suite of hand-crafted edge cases was implemented independently of the 10,000-file run. These tests targeted specific vulnerabilities in schema inference and execution.

Key edge cases addressed:

- Missing or default headers: Auto-renamed based on first-row values (e.g., `col_name`, `col_value`)
- Whitespace normalization: Leading/trailing spaces stripped from headers
- Quoted fields with internal delimiters: Handled via CSV dialect-aware parsing
- Type inference: Based on pandas' dtypes mapped to Jayvee types (boolean, integer, decimal, text); datetimes coerced to text
- Header repair propagation: Renamed headers dynamically propagated to downstream blocks
- Boolean type detection: Used pandas-native logic (without extended normalization)
- File/path normalization: Filenames auto-normalized to CamelCase block identifiers

Additionally, runtime logging captured read/write errors with timestamps, ensuring issues could be traced without halting the batch process.

Some complex challenges were identified but remain unimplemented in this iteration:

- Unicode normalization for multilingual headers
- Robust date coercion beyond ISO formats

- Semantic handling of diverse null patterns

These edge-case tests, though limited in scope compared to the full 10,000 pipeline runs, demonstrate the system's flexibility and provide a foundation for future enhancements in schema inference under non-ideal input conditions.

6.2 LLM Schema Inference Evaluation

The large-scale evaluation setup for header-row inference using transformer models under SLURM orchestration is detailed here.

6.2.1 Evaluation Workflow

The evaluation framework for Jayvee's schema inference module was designed to rigorously test the model's ability to identify header rows in CSV files that reflect real-world messiness, ambiguity, and inconsistency. Rather than focusing on architectural constraints, the design centers on controlled variation, scalability, and error surface visibility—all essential to understanding the strengths and limitations of LLM-driven schema detection.

To generate a structurally diverse and realistic set of test CSV files for evaluation, synthetic data was created using ChatGPT as a controlled generative source. The generation process was guided by prompt engineering: ChatGPT was explicitly instructed to simulate CSV files that reflect a wide variety of noise patterns and formatting irregularities typically found in real-world datasets. These prompts included multiple example CSVs containing preambles, comment lines, mixed delimiters, unit annotations, and irregular metadata blocks, which helped steer the model toward producing files that mirrored the complexity of practical data ingestion scenarios.

One key constraint enforced during generation was that the actual header row should always appear within the first 10 lines of each file. This constraint reflects the empirical

observation that, in most practical datasets, structural metadata or introductory content rarely exceeds this threshold before the tabular schema begins. It also helped establish a consistent search window for evaluation, allowing models to focus on early sections of the file without assuming unlimited context.

To validate the fidelity and diversity of the generated files, a manual spot-checking procedure was employed. Random samples from the 10,000-file corpus were reviewed to verify that they met key criteria: the presence of varying forms of noise (such as comment blocks, unit rows, encoding hints), differences in header positioning within the allowed range, and legitimate variability in schema complexity. This qualitative inspection confirmed that the files spanned a representative spectrum of realistic CSV challenges and that the header annotations aligned with plausible human expectations.

Overall, this methodology enabled the creation of a high-variance synthetic benchmark that effectively stresses header detection models across a wide range of structural ambiguity, without requiring extensive hand-labeling. The controlled use of ChatGPT for generation, combined with spot validation, provided both scale and quality in dataset preparation.

Every file is paired with a ground truth label specifying the correct header row, stored in a separate JSON file. This design ensures high-throughput testing while enabling exact comparisons between predicted and expected schema locations.

To operationalize this testing benchmark, the system employs a SLURM-based batch processing design. Once launched, the master script configures the compute environment, loads necessary modules, activates the relevant Conda environment, and sets up directories for logs, inputs, outputs, and metrics. Each CSV is then passed through a uniform evaluation harness that locally loads the Hugging Face model, constructs a prompt using the same few-shot template as production inference, and performs local, deterministic generation using the transformers library. Each model is launched in a separate SLURM array task (parallel_evaluation.sh) and evaluated independently for fair comparison.

Models output a structured JSON with two keys: `columnNameRow` (integer) and `Explanation` (string). The former is compared against ground truth to assess accuracy.

44

The model's output is parsed, validated, and compared against the known header index. This comparison feeds directly into an aggregated accuracy score. For each file, detailed logs are captured—including the full prompt, model output, any exceptions raised, and metadata about runtime behavior. This transparent logging framework supports prompt audits, error tracing, and cross-run reproducibility.

The script `evaluate.py` coordinates evaluation per model, while `aggregate_results.py` consolidates all partial evaluations into final JSON outputs. This two-stage design decouples inference from aggregation, allowing partial results to be recovered and reused if jobs fail mid-run.

What distinguishes this implementation is its emphasis on semantic difficulty over syntactic cleanliness. The diversity and density of obfuscating preamble content test not just the model's token-level pattern recognition, but its higher-order reasoning about what defines a schema. Rather than simply identifying the most densely delimited or capitalized row, models must interpret intent, context, and structure—skills central to robust schema understanding.

6.2.2 Evaluation Results

A synthetic benchmark of 10,000 structurally diverse CSV files was used to evaluate three transformer-based language models on their ability to correctly identify header rows. Each prediction was considered accurate only when it precisely matched the annotated ground truth. The benchmark deliberately included edge cases such as lengthy multi-line preambles, non-standard delimiters, multilingual comments, and hybrid metadata patterns to rigorously test the models' semantic understanding.

Performance across the three models revealed substantial variation, shaped by differences in parameter size, instruction tuning, and schema inference capabilities. The instruction-tuned DeepSeek-Coder model achieved the highest accuracy, with 85% of headers identified correctly. It exhibited strong generalization across noisy and irregular file structures, showing resilience to comment-laden preambles and embedded metadata.

45

In contrast, Qwen3-4B, a smaller model with more limited schema-specific fine-tuning, reached around 72–75% accuracy. While it performed reliably on simple or moderately obfuscated headers, its performance degraded when encountering files with complex metadata layering or pre-header unit annotations. Surprisingly, CodeLlama-7B-Instruct, despite having more parameters than Qwen3-4B, delivered less consistent results, hovering around 65–68% accuracy. It often prioritized superficial text features like capitalization or row length over semantic cues, and frequently misclassified visually dense rows without adequately interpreting their functional roles.

An in-depth error analysis revealed several common failure modes. Models frequently mistook rows containing unit descriptions—such as "All values in USD"—for actual headers, particularly when those lines were short, capitalized, and well-formatted. This tendency indicates an overreliance on surface-level heuristics in the absence of semantic understanding. Another recurring issue was the misidentification of document footers or introductory comments as header lines, pointing to the models' limited ability to temporally contextualize structural elements within the typical progression of a CSV file. The inability to resolve multi-row headers further exposed a structural weakness: rather than aggregating schema information spread across adjacent lines, models typically fixated on a single row, leading to partial or incorrect extractions. Lastly, all three models demonstrated a strong bias toward visually regular patterns—favoring consistent delimiters, token length, and uppercase tokens—even when such rows carried no semantic weight as headers.

These patterns suggest that accurate schema inference remains a partially emergent capability among general-purpose LLMs. Instruction tuning—exemplified by the superior performance of DeepSeek-Coder—clearly enhances robustness and contextual understanding, but persistent failure modes underscore the absence of domain-specific inductive bias and the lack of hierarchical reasoning. Notably, the results also challenge the assumption that larger models inherently perform better on structural tasks; CodeLlama's comparatively weak results demonstrate that parameter count without targeted training does not guarantee semantic precision.

From a practical standpoint, these findings have direct implications for deploying LLMs in data ingestion and integration pipelines. In production environments, where files often contain noise, metadata, and domain-specific annotations, relying solely on LLM predictions without safeguards may lead to brittle performance. Incorporating fallback strategies—such as rule-based validation layers, confidence-based filtering, or hybrid pipelines combining symbolic and neural methods—could mitigate these risks. Additionally, fine-tuning on curated corpora of annotated, messy CSVs could further align model predictions with real-world schema patterns.

The experimental setup itself, grounded in synthetic data with controlled complexity and transparent, traceable evaluation criteria, provides a solid foundation for future benchmarking and iterative refinement. It not only demonstrates the current feasibility of LLM-based header detection but also delineates the performance ceiling and the kinds of architectural or training interventions that may be necessary to push it further.

7 Conclusions

7.1 Template Generation

The Jayvee template generation pipeline has matured into a robust, modular system capable of transforming diverse and often messy CSV schemas into executable Jayveepipelines at scale. The updated batch-driven architecture, powered by a SLURM-managed HPC setup, successfully processed 10,000 files without manual intervention and maintained sub-second generation latency per file.

Key robustness features—such as fallback renaming of unnamed columns, whitespace trimming, quoted field handling, and consistent CamelCase pipeline naming—ensure that input irregularities do not propagate downstream. Data type inference via pandas provides effective generalization across boolean, numeric, and text types, though datetime normalization remains a known limitation.

The system's deterministic, file-based I/O and centralized logging strategy has enabled full automation, observability, and testability. While complex cases like multilingual headers and semantic null handling are not yet supported, the current implementation offers a strong foundation for scaling and iterative enhancement. Future iterations could integrate plugin support for custom block types and improve remote URL handling for better security and extensibility.

7.2 LLM Schema Inference

The LLM-based schema inference module demonstrates the viability of using transformer models for detecting CSV header rows under real-world obfuscation. Leveraging a

49

synthetic benchmark of 10,000 noisy, variably structured files, the evaluation framework revealed critical differences in accuracy, robustness, and reasoning depth across models.

The instruction-tuned DeepSeek-Coder model performed best, highlighting the benefits of task-specific tuning. Smaller or less specialized models, such as Qwen3-4B and CodeLlama-7B-Instruct, showed sensitivity to superficial token patterns and struggled with hybrid or metadata-laden preambles. These findings affirm that header detection is a semantic task requiring more than delimiter detection or capitalization heuristics.

The evaluation harness—modular, traceable, and SLURM-parallelized—supports reproducible benchmarking and detailed error analysis. However, reliance on local GPU-backed inference constrains scalability and limits accessibility in low-resource settings. Furthermore, the non-determinism of generative outputs complicates reproducibility, suggesting the value of prompt freezing, scoring ensembles, or hybrid approaches combining LLMs with regex-based heuristics.

8 Future Work

8.1 Template Generation System

Enhancing the Jayvee template generation engine includes both standalone improvements and tighter integration into broader data infrastructure—especially the JValue Hub ecosystem.

A high-priority direction is embedding the template generator as a native feature in JValue Hub. This would allow users to select a CSV file or remote data source via the Hub's UI and automatically receive a pre-generated .jv pipeline—complete with inferred schema, ready-to-run transformations, and downstream sinks. This "one-click pipeline generation" experience would empower non-technical users and accelerate onboarding in data-centric workflows. Internally, the generator can be wrapped as a web API microservice callable from the Hub's backend, preserving modularity and enabling scalable deployment.

Beyond platform integration, the core generator itself can be improved in several ways. Migrating the system into a lightweight web service would provide broader deployment flexibility and facilitate use in automated workflows or CI/CD environments. Introducing a plugin framework would allow users to define custom extractors, converters, and sinks without modifying the core logic, increasing extensibility for domain-specific applications. Performance can be further optimized by profiling and parallelizing the pandas-based type inference and serialization logic, which is especially valuable for handling large or high-frequency datasets. Additionally, strengthening the security model —such as sandboxing remote file access, validating input schemas, and defining fetch policies—will be essential for safe adoption in enterprise and cloud-based settings.

To increase versatility, the generator should support multilingual column header normalization and be fully Unicode-aware to accommodate international datasets. Allowing

for configurable coercion strategies, particularly for temporal, boolean, or nullable fields, would provide more control over schema fidelity. Finally, enabling real-time validation and preview of inferred pipelines within interactive environments—whether via notebook, IDE extension, or web frontend—would improve user trust and rapid feedback during the authoring process.

By combining modular extensibility with seamless platform integration, the template generation tool can evolve from a developer-centric utility into a foundational automation layer within JValue Hub, reinforcing the platform's vision of composable and declarative data workflows.

8.2 LLM Schema Inference Module

The LLM-based schema inference module remains fertile ground for innovation, with priorities centered on improving accuracy, robustness, and scalability.

To improve prediction quality, future work should explore retrieval-augmented generation and adapter-based fine-tuning using a curated corpus of annotated CSVs. These techniques could significantly reduce hallucinations and improve semantic consistency. Incorporating heuristic fallbacks will further stabilize inference in edge cases, such as files with embedded units, metadata rows, or irregular formatting. Expanding the prompt design to support multilingual header interpretations will enhance generalization across non-English and locale-specific datasets.

From a systems perspective, efficiency and portability can be improved by developing distilled or quantized variants of the best-performing models. Such variants would enable inference in resource-constrained environments, including browser-based clients or serverless runtimes. It may also be beneficial to implement dynamic model selection logic, allowing the system to switch between lightweight and heavyweight models depending on file complexity, size, or domain metadata.

Employing checkpointing mechanisms and caching partial results will reduce recomputation during long-running batch jobs and make iterative runs more efficient. In tandem, expanding the benchmark suite to cover diverse file formats such as TSVs, XLSX files, and fixed-width tables, along with scraped or non-standard real-world examples, will ensure generalization and provide a more rigorous evaluation baseline.

Ultimately, integrating the schema inference module directly into JValue Hub as a schema-suggestion feature would complement the template generator. Together, they would form a seamless "infer + generate" workflow, substantially lowering the barrier from raw data to validated, runnable pipelines and making the entire process accessible to a broader audience.

Bibliography

[JVN21]   L. Jiang, G. Vitagliano, and F. Naumann, "Structure Detection in Verbose CSV Files," in *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2021, pp. 193–204.

[Sui+24] Y. Sui *et al.*, "TAP4LLM: Table provider on sampling, augmenting, and packing semi-structured data for large language model reasoning," *arXiv preprint*, 2024, doi: 10.48550/arXiv.2405.14228.

[Liu+24] L. Liu *et al.*, "AutoDW: Automatic data wrangling leveraging large language models," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024)*, 2024. doi: 10.1145/3691620.3695267.

[Guo+24] D. Guo *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence," *arXiv preprint*, 2024, [Online]. Available: https://arxiv.org/abs/2401.14196

[Yan+25] A. Yang *et al.*, "Qwen3 Technical Report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[Roz+23] B. Rozière *et al.*, "Code Llama: Open Foundation Models for Code." [Online]. Available: https://arxiv.org/abs/2308.12950

[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: 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," *arXiv*, 2025, [Online]. Available: https://arxiv.org/abs/2310.10508

[Zha+24] X. Zhang *et al.*, "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, 2024.

[SPA23] B. Smock, R. Pesala, and R. Abraham, "Aligning benchmark datasets for table structure recognition," in *International Conference on Document Analysis and Recognition (ICDAR)*, Mar. 2023, pp. 371–386. doi: 10.48550/arXiv.2303.00716.

[Shi+23] J. Shin, C. Tang, T. Mohati, M. Nayebi, S. Wang, and H. Hemmati, "Prompt engineering or fine-tuning: An empirical assessment of large language models for code," *arXiv preprint*, 2023, [Online]. Available: https://doi.org/10.48550/arXiv.2310.10508

[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," Manchester, UK, 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, A. Ritter, and others, "Schema-driven information extraction from heterogeneous tables," in *Findings of the Association for Computational Linguistics: EMNLP*, 2024, pp. 10252–10273. doi: 10.18653/v1/2024.findings-emnlp.600.

[WCP25]   Z. Wu, J. Chen, and N. W. Paton, "Taxonomy inference for tabular data using large language models," in *The Extended Semantic Web Conference (ESWC)*, 2025. doi: 10.48550/arXiv.2503.21810.

[Dee24]   DeepSeek-AI, "DeepSeek Coder: Retrieval-Augmented Large Language Model for Tabular Data Understanding." [Online]. Available: https://arxiv.org/abs/2501.12948

[Qwe24]   QwenLM, "Qwen3-4B: A Multilingual Instruction-Following Model." [Online]. Available: https://huggingface.co/Qwen/Qwen3-4B

[Met24]   Meta AI, "CodeLlama-7B-Instruct: Instruction-Tuned Code Generation Model." [Online]. Available: https://huggingface.co/codellama/CodeLlama-7b-Instruct-hf

[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. Van 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 Repre-*

sentations (ICLR), 2020. [Online]. Available: https://openreview.net/forum?id=HkejNgBtPB

Acronyms

CLI Command-Line Interface

DSL domain-specific language

ETL Extract-Transform-Load

LLM Large Language Model

PLDI Programming Language Design and Implementation

RAG retrieval-augmented generation

RALM Retrieval-augmented language models

RLHF reinforcement learning with human feedback

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:

- Jayvee version: 0.6.4
- node.js version: v22.16.0
- 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`

LLM-Based Schema Inference

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

Notes

- The listed Python packages are version-pinned to guarantee reproducibility and cross-platform compatibility.
- The development and testing environment was verified on MacOS and Linux.