Universiteit Leiden

Master Computer Science

LLM-DSF: Automating data science by harnessing the power of Large Language Models

Name:	Remco Stuij
Student ID:	S3820408
Date:	28/04/2026
Specialisation:	Data Science
Daily supervisor:	A. Paraskeva
1st supervisor:	J. N. van Rijn
2nd supervisor:	S. Verberne

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Einsteinweg 55
2333 CC Leiden
The Netherlands

ABSTRACT

End-to-end data science is labor-intensive, especially in upstream tasks such as data inspection, cleaning, and feature engineering. This thesis introduces LLM-DSF, a modular framework that turns a task prompt and tabular data into an executable Python program. LLM-DSF preprocesses and splits the data, derives metadata, builds a structured prompt, extracts code from the model response, and executes the results inside an isolated container. The framework can retry generations when errors occur or apply feedback to enable iterative improvement. LLM-DSF supports both local .gguf models and API backends, such as OpenAI and Gemini. Experiments indicate that metadata and the error-retry loop significantly increase the success rate. Across both public and private datasets, LLM-DSF improves robustness compared to the direct cloud LLM variants. However, the main limitation is that LLM-DSF does not acquire the predictive performance of specialized AutoML systems. Overall, LLM-DSF is best positioned as a versatile automation tool for generating reproducible baselines and end-to-end programs, rather than a replacement for performance-focused tools or humans.

CONTENTS

1 INTRODUCTION

Across industry, government, and science, organizations increasingly rely on data-driven decision-making. This shift results in a significant increase in demand for end-to-end data science, transforming raw data into models that support prediction, explanation, and operational decision-making. However, the complete process is time-consuming, labor-intensive, and iterative. In particular, a large share of the required time is spent before model training even starts, on diagnosing data quality issues, transforming tables, and iteratively refining features [10].

The need for end-to-end data science pipelines has driven a surge in demand for data scientists to keep pace with the work. The U.S. Bureau of Labor Statistics projects a 34% increase in data science job openings, supporting this demand [4]. In the UK, similar evidence shows that the supply of data specialists does not meet the rising demand, 62% of the respondents reported that they could not achieve their data goals due to a lack of specialized knowledge among both job applicants and existing staff [14]. The trends highlight a consistent problem: many organizations want to use data, but the expertise and time needed remain scarce.

Automating parts of the end-to-end data science lifecycle has been a research focus for over two decades. Still, it gained real traction in the 2010s with systems that operationalize automated machine learning (AutoML), such as AutoSklearn [9]. Most AutoML systems explain model development as the combined algorithm selection and hyperparameter problem. They use search and meta-learning to reuse experience from prior datasets, thereby reducing manual trial-and-error and producing strong models with limited expert input [1]. AutoML typically assumes that the prediction target, an evaluation metric, and usable tabular data already exist, leaving much of the practical labor of data science pipelines untouched. This gap motivates methods that push automation upstream in the workflow, before model selection and tuning [3]. De Bie et al. further explore that gap, claiming that the part of data science that is not automated accounts for over 80% of the time required, and upstream steps such as data exploration, preparation, and feature engineering still require human expertise [2].

Recent advances in large language models (LLMs) offer a new approach to addressing the upstream automation gap. Following the introduction of the transformer architecture [24], LLMs are trained on large-scale text corpora to learn the general representations of language, followed by LLMs trained on code bases to translate natural language instructions into executable programs [6]. Over the years, models have been significantly improved in the race to achieve the best coding performance, enabling users to generate code for tasks such as data cleaning, transformation, and feature construction. This capability suggests that LLMs can complement AutoML by targeting upstream tasks where the gap in AutoML arises. FunSearch further illustrates that LLM-generated programs become more useful when they are combined with systematic evaluation and iteration [20], although the focus here lies on mathematical discovery rather than data science automation, its generate, evaluate, and improve structure supports the relevance of feedback-driven LLM frameworks for producing executable and assessable code.

This thesis investigates how LLMs can be integrated into a practical, reproducible, and safe framework for automating data science work on tabular data. I introduce the Large Language Model Data Science Framework (LLM-DSF),[1] a modular system that couples LLM-driven code generation with deterministic execution and structured feedback. Given a task description and a dataset, LLM-DSF (i) preprocesses the dataset and constructs an LLM-friendly representation, (ii) prompts an LLM to generate an end-to-end Python solution, and (iii) executes the generated code in an isolated containerized environment. The framework then returns the program's output and errors and can feed them back to the model to support iterative improvement. LLM-DSF aims to jump-start projects for data scientists while keeping the user in control of the goals, constraints, and the final model.

This thesis addresses the central question:
How can LLM-DSF be constructed to autonomously generate high-quality models?
To address this, three guiding sub-questions are proposed:

1. **What core design principles and mechanisms support an LLM framework to autonomously generate code for various stages of the data science lifecycle?**

   This sub-question identifies the essential design elements needed for iterative automation of data science tasks.

2. **What standards should define "high quality" in the context of code and models generated by an LLM framework for data science processes?**

[1] https://github.com/Remmie0/LLM-DSF

This question establishes quality benchmarks, ensuring reliable and practically useful outputs and adding measurability to the proposed method.

3. **How can the framework incorporate safety and assessment mechanisms to consistently evaluate and adapt its outputs to meet high-quality standards across the data science lifecycle?**
   This sub-question examines how adaptive processes can help maintain output quality throughout the data science pipeline.

Scientific contributions. This research makes three main contributions:

1. It presents LLM-DSF, a modular framework for automating end-to-end data science via code generation using LLMs.
2. The research designs a controlled workflow around the LLM that combines deterministic preprocessing, structured prompting, isolated code execution, and feedback loops to improve the safety, reproducibility, and robustness of the framework.
3. It assesses this design on both public and private datasets, showing that robustness improves with the provision of metadata and the use of error-retry loops, although predictive performance remains below that of specialized AutoML systems.

2 Related work

The related work section reviews literature on papers that focus on (i) mechanisms for end-to-end automation with limited human intervention, (ii) how prior work aims for "the high quality" for generated code and models, and (iii) how systems assess and control risks when they execute code-generated programs. The selection is made to answer the three sub-questions from section 1.

2.1 Automation mechanisms across the data science lifecycle

Automated machine learning (AutoML) is a common starting point for automation that focuses on the modeling stage, assuming that the data is analysis-ready. Auto-sklearn formalizes model development by combining model selection, hyperparameter optimization, and ensemble models using automated search and meta-learning to construct pipelines with limited expert input [9]. More broadly, meta-learning studies show how to reuse experience from previous datasets to accelerate search and improve performance on new tasks [3]. To compare AutoML systems, benchmark suites provided a standardized setting with shared datasets, evaluation metrics, and runtime budgets. One such example is the AutoML Benchmark, which is designed to support systematic comparison of AutoML frameworks under controlled constraints [11].

The emphasis of AutoML on the modeling stages leaves work in the upstream stages, where data specialists still write task-specific scripts for data inspection, cleaning, transformation, and feature engineering. Prior research describes this mismatch as an open challenge, because upstream steps are diverse, context-dependent, and often require domain knowledge [2]. Work on data wrangling similarly highlights that preparing data involves operations, such as resolving missing values, standardizing formats, and reshaping tables, and therefore does not fit a single optimization objective, as hyperparameter tuning often does [10]. Taken together, these motivate automation mechanisms that extend beyond the traditional "choose and tune a model" paradigm and support iterative data preparation and reasoning about data constraints.

Implications for Sub-question 1. Related literature has shown that an autonomous framework must manage multiple stages from data preparation to evaluation and must support iteration. AutoML provides mechanisms for systematic search under constraints, whereas end-to-end automation requires context modeling and flexible representations for both the data and the goal.

2.2 LLM-driven automation for data science tasks

LLMs trained on code provide a different approach to automation than classical AutoML; instead of searching a predefined space, they generate executable programs from natural-language instructions. With the invention of transformer-based language modeling [24], and large-scale code training enables LLMS to map task descriptions to scripts that implement data preparation, modeling, and evaluation [6]. This shifts part of the automation problem from designing a search space to designing an interaction loop that supplies the model with sufficient context, translates the response into runnable code, and handles failure cases.

Several recent works translate this idea to the context of data science. AutoDW targets data wrangling and uses LLMs to automate transformations that would otherwise be implemented manually, serving as a generator for cleaning and restructuring code [17]. DS-agent proposes a different LLM use case by employing case-based reasoning to adapt solutions from prior examples to new problems, aiming to improve output reliability [13]. Beyond data science, FunSearch is developed with a general pattern for program generation: it generates candidate solutions, externally evaluates them, and feeds the evaluation signal back into the search process [20]. This generate-evaluate loop is interesting, as many data science tasks follow the same pattern: generating code and evaluating it against metrics, runtime success, or constraint violations.

A core challenge in LLM-driven automation for tabular data is representing the input in a way that supports correct reasoning. Table Meets LLM studies how LLMs handle structured tables and provides evidence that the interface between structured data and language models affects performance, motivating choices in how tabular data is presented in prompting [21].

Implications for sub-question 1. Across LLM-based systems, two mechanisms are needed: first, an explicit interface layer that translates structured datasets to textual model inputs. Secondly, iterative

refinement, in which the generated code is coupled to evaluation signals such as execution results or errors.

2.3 Quality standards for generated code and models

The second sub-question aims to more precisely define the term "high-quality" in the context of code and model generation from an LLM-driven framework. Quality can be measured on multiple levels, namely (i) the program level, (ii) the pipeline level, and (iii) the model level.

Program-level quality. At the program level, it is at a bare minimum that code must be functionally correct, meaning the code must execute successfully in the intended environment and produce outputs of the expected type. In practice, this would entail handling dependencies, appropriate file access patterns, and consistent use of data schemas. However, because LLMs generate free-form code, runtime failures are common, which motivates evaluations that record execution success and provide feedback whenever errors happen [6].

Pipeline-level quality. The pipeline level shows whether or not the program follows the accepted methodological steps. Prior research emphasized that data science involves iterative problem understanding, data preparation, modeling, and evaluation, and that these steps should be reliable and scientifically reproducible [2]. For tabular data, pipeline-level validity often depends on correct train/test separation, prevention of leakage, and appropriate task selection. The evaluation here is not "did the code run?" but "did the code implement a solution that is methodologically sound?".

Model-level quality. The deepest level is the model-level quality, typically defined by generalization performance on held-out data, using appropriate metrics. AutoML research has established standardized protocols that compare model quality across datasets [9, 11]. These protocols clarify that model quality is defined by performance, reproducibility, and fairness of the evaluation setting.

Benchmarking LLM agents for data science. Benchmarks for LLM agents extend this evaluation perspective to LLM-driven tasks. DataSciBench proposes a benchmark to evaluate LLM agents on data science tasks, which supports not only assessing the final results, but also whether the agent reliably completes all required steps [25].

Implications for sub-question 2. Literature suggests that "high quality" should not be defined as one metric, but rather on multiple levels: (i) program correctness, (ii) pipeline validity, and (iii) model quality. This motivates evaluation mechanisms that operate on these different levels.

2.4 Safety and assessment mechanism for autonomous execution

LLM-driven automation can produce both useful and incorrect or inappropriate code for a given task. Tornede et al. discuss both the risks and opportunities of combining AutoML with LLMs, where reliability issues such as hallucination or unclear assumptions lead to the need for robust evaluation and guardrails for LLMs to be used in optimization loops [22].

A recurring mechanism in the literature for improving reliability is the provision of external feedback to the program. In generated settings, feedback can be derived from runtime errors, evaluation metrics, or constraint checks. DS-Agent uses case-based reasoning to guide generation toward known patterns, thereby improving reliability [13]. FunSearch uses an even stronger form of feedback by using automated evaluators to guide the program to better objective values [20]. AutoDW applies LLM generation to data wrangling tasks, where correctness can often be checked by validating the transformations [17]. All of these methods share the idea that safety and quality control require explicit external signals, rather than relying solely on the model's confidence.

A second recurring mechanism is to control the execution environment and permissions when running generated code. Although many papers focus on modeling and evaluation rather than on systems security, it has been observed that practical frameworks require isolation, resource limits, and auditable logging to enable automated execution. For example, FunSearch executes programs in isolated containers with limited system access to control the effect of the generated code [20].

Implications for sub-question 3. It is clear that assessment methods should be external and explicit. The most common signals are (i) runtime feedback, (ii) task metrics on held-out data, and (iii) constraint checks that detect invalid steps.

2.5 Summary and research gap

AutoML research provides strong baselines and evaluation for the modeling stage, but assumes that data is already prepared for learning [9, 11]. End-to-end data science automation remains difficult and requires iterative decision making [2]. LLM-based systems show that code generation and feedback loops are promising mechanisms to address this gap [6, 20]. That said, the literature shows that quality and safety require explicit evaluation signals to function properly [25, 22]. This thesis builds on these insights by proposing a framework that empirically assesses which systems most affect robustness and usefulness across the entire data science lifecycle by integrating elements from different papers into a single system.

3 METHODOLOGY

The framework's methodology will be divided into several sections. First, an overview will be provided in section 3.1. Parsing and processing the input will follow in section 3.2. Instantiation and generation of the large language model will be explained in section 3.3, combined with parsing the response in section 3.4. followed by running the code in containers in section 3.5 and the implementation of the feedback mechanisms in section 3.6. Finally, this chapter will close with the implementation of a test suite for a robust framework in section 3.7.

3.1 ARCHITECTURAL OVERVIEW

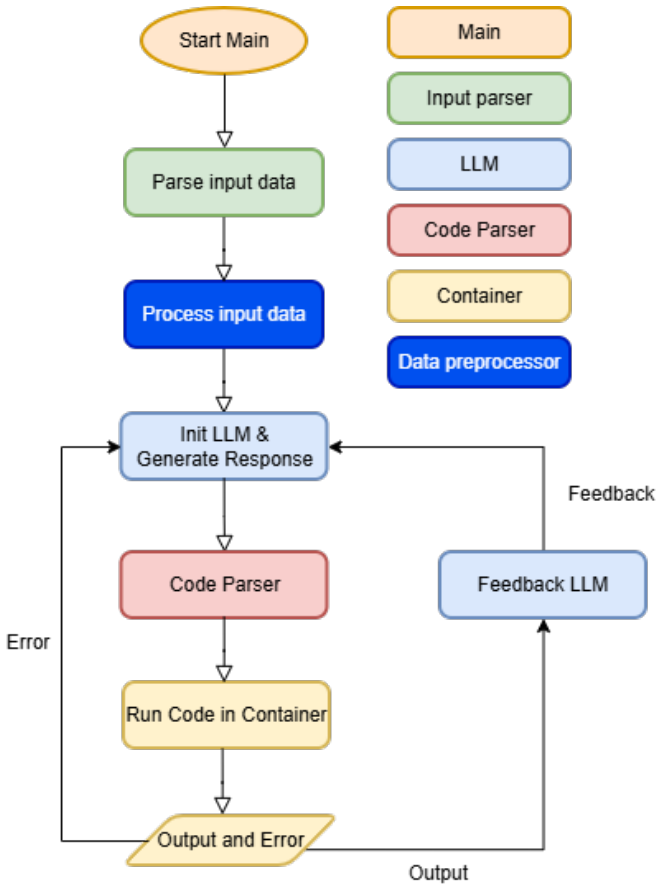

Figure 1: The overview of the architectural diagram of the LLM-DSF framework.

I implement the LLM-DSF framework during this research. The implementation includes the input parser, data preprocessor, prompt construction, model interface, code extraction, feedback and error-retry loops, and containerized execution. The framework builds on existing Python libraries for data handling, model inference, and containerization, but the orchestration and integration of these components are within the scope of this research.

Figure 1 provides a general overview of the control flow of the LLM-DSF framework. The program starts with input data, a task prompt, or both. The input parser then parses these inputs, after which the raw data is sent to the data preprocessor in tabular format. The preprocessor standardizes column names, splits the data into training and test sets, and, when metadata extraction is enabled, derives metadata from the training set to feed to the LLM. The prompt is then being constructed by concatenating the task instructions with a sample of the training data and, if enabled, the training-derived metadata.

The LLM will provide a natural-language response that must be parsed to obtain an executable program. To improve reproducibility and security, the framework executes the code in an isolated container with read-only access to the data. The execution results either in (i) program output or (ii) a runtime error.

If execution fails, the framework optionally enters an error loop, constructs a corrective prompt that includes the previous code and the error, and retries generation for a fixed number of attempts. If execution succeeds, the framework optionally enters a feedback loop, in which a new LLM instance is used to generate structured feedback. This structured feedback is then incorporated into a new prompt to the original LLM to try to improve the solution. All prompts, code, outputs, and errors are logged for inspection.

3.2 Process input

Processing the input is divided into two parts: first, parsing the input data by reading the prompt and data files in section 3.2.1. Followed by the data being processed, which is explained in section 3.2.2.

3.2.1 Process input - parsing

The framework requires a prompt in text format and a data file, which can be either a data format or an OpenML task ID [8]. The prompt is read to obtain the task prompt, which is then provided to the LLM.

The data file is handled across several formats and currently supports 10 file types, which are read using a mapping from file extensions to the corresponding pandas functions, as shown in Table 1. The pandas table is then passed to the data processor.

File extension(s)	Pandas reader used
`.csv`	`pd.read_csv`
`.tsv, .tab`	`pd.read_csv(sep=\t)`
`.json`	`pd.read_json`
`.xlsx, .xls`	`pd.read_excel`
`.parquet`	`pd.read_parquet`
`.feather`	`pd.read_feather`
`.pkl`	`pd.read_pickle`
`.h5`	`pd.read_hdf`
`.orc`	`pd.read_orc`

Table 1: Supported input file types and the pandas readers used by the framework's input parser.

Alternatively, the data can be provided to the framework as an OpenML task ID. The framework retrieves the task using the provided task ID and creates a raw table, which is then passed to the data processor.

3.2.2 Process input - processing

The related work section 2.2 shows how LLMs perform with structured data is largely affected by how the data is provided to the model. LLM-DSF leverages lessons learned from prior work by adequately preprocessing the data [10, 21].

The data is preprocessed in multiple steps:

- Sanitizing column names
- Splitting the data
- Generating metadata
- Providing Sample rows

Sanitizing column names. Sanitizing column names makes it easier for large language models to generate code that references specific column names. The sanitizing is being done by:

- Make all characters lowercase
- Replacing all spaces with _
- Replacing all special characters with _
- Remove all leading and trailing _

This yields normalized tables that are more easily interpreted by LLMs and improve the stability of the code generated by LLMs.

Splitting the data. The data processor splits the normalized tables into two sets: training and test. This is done for subsequent processing steps, as the model should have access only to metadata and sample rows from the training dataset to prevent data leakage during generation.

Generating Metadata. The preprocessor generates a table schema that shows, for each column, its data format, number of unique values, and, for numeric data types, summary statistics. The summary statistics generated are the mean and range, and provide some background information for the LLM to work with. Finally, the dataframe shape is calculated and added. All metadata is generated from the training dataset, thereby preventing data snooping. An example of the generated metadata is shown in Appendix B, figure 5.

Providing Sample Rows. The last thing the data processor is doing in the framework is randomly selecting a number of sample rows from the training dataset and concatenating all the values with pipe operators, such that the LLM can easily identify the values. The number of sample rows is a user-defined parameter defaulting to 5. If the number of sample rows is larger than the total length of the training dataset, then all the rows are added to the prompt. An example of the sample rows added to the prompt is shown in Appendix B, figure 6. The DeepSeek models support up to 128.000 tokens as a context window, but due to hardware limitations, a maximum of 16.384 tokens is used as input [12].

3.3 Large language model

The large language model (LLM) component is the central decision mechanism of the LLM-DSF framework. The framework implements a single interface that (i) constructs the final prompt from the task instructions and preprocessed data, and (ii) generates a response using a selected model back end. This separation enables controlled experimentation because prompt structure and generation parameters can be scaled independently of the execution environment.

3.3.1 Large language model - model back-end abstraction

LLM-DSF supports three types of model back ends:

- local quantized models in .gguf format
- the OpenAI Api
- the Google Gemini API

The framework selects the backend via the 'model_type' parameter in the LLM interface. For the .gguf models, the framework initializes a local model object and performs the model inference locally. For both API-based models, the framework instantiates a provider client (OpenAI) or creates a client during prompting (Gemini) and sends the prompt to the configured model.

During the development of this framework initially, three distilled models of DeepSeek-R1 have been used. During one of the experiments, it was found that the Qwen distilled models performed way more consistently than the Llama distilled DeepSeek models in section 4.2 [12]. Finally, the default model used during the project was 'DeepSeek-R1-Distill-Qwen-32B-Q4_K_M'. [2] Note that local model selection is out of scope for this project's research and can be addressed in future work.

3.3.2 Large language model - prompt building

Prompt building in the framework combines up to three inputs into one final prompt:

- the task prompt provided in .txt format
- sampled data from the training dataset
- optionally metadata calculated over the training data (see section 3.2.2)

The framework appends the metadata when configured as a 'Data Structure' block and the sampled rows as a 'Data Sample' block. This structure standardizes the prompt layout across tasks and enables ablation studies to find the effects of including metadata.

For API-based models, the framework also uses a system-level instruction string that constrains the output format.

[2] https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-32B

3.3.3 Large language model - response generation

Response generation takes the final prompt and requests a response from the selected back-end model. The framework exposes parameters for local inference, such as 'temperature', which can be tuned to control model behavior, constrain the model's output, or offload tasks to the GPU.

The default values will be set to:

- temperature = 0.2 (see section 4.2)
- n_ctx = 16392 (due to hardware limitations)

API responses will be kept at the API provider's defaults.

3.4 Code parser

The large language model's response is text, whereas the container must contain valid Python code. LLM-DSF therefore applies a parsing step that extracts code blocks from the response. Note that models are always instructed to return the code in a single final Python cell block to improve the likelihood of finding the code in the response.

The parser searches for fenced code blocks in Markdown format. It uses a hierarchical ordering for extracting the code from the response. First, it looks for a specific Python Markdown block in the following format: ``` ```python ... ``` ```. If that does not exist, it falls back to generic Markdown cell blocks in the format: ``` ```... ``` ```. Finally, if multiple blocks exist, only the last block is selected, as it is most likely to contain the final answer, given the model's reasoning. If no code block is found, the parser returns nothing, which triggers the appropriate error handling in the feedback mechanism.

Experiments using a simple prompt to generate code for the Fibonacci sequence and return the first 5 numbers show that out of 800 trials, only 5 lacked a valid code block, and the parser found code in the remaining 795 cases.

3.5 Containerization

The LLM-generated code usually executes on the host system with the same privileges if executed directly. LLM-DSF mitigates this risk and improves reproducibility by running the generated code inside a Docker container.

The framework passes the preprocessed dataset from the input process to the container by temporarily transforming it into a .parquet file and mounting the temporary drive as a read-only volume inside the container. The LLM-DSF modifies the code to either read the dataframe from the temporary volume or load the data from OpenML. This design ensures that the executed code does not depend on file paths that may differ between runs or systems. Examples can be found in Appendix C, generated pre-modified code in figure 7, and the LLM-DSF modified code in figure 8.

The container environment captures stdout and stderr separately and returns both to the orchestration process. Execution is also bound by a timeout mechanism to prevent a container from being stuck in code that takes too long; in this research, the timeout was set to 30 minutes. LLM-DSF automatically removes the container and temporary directories after execution to avoid resource leakage.

The container environment provides two benefits to the overall framework. First, it is a safety mechanism; code is executed in an isolated read-only data environment. Second, it improves reproducibility as the runtime dependencies and the container image are fixed and applied to every generated program.

3.6 Feedback mechanism

LLM-DSF implements an optional dual-feedback mechanism that refines model outputs: an error-correction loop and a quality-improvement feedback loop. Both loops are bounded by a fixed maximum depth, controlled by parameters provided in the code or via the command-line interface, to ensure termination.

3.6.1 Feedback mechanism - error loop

The error-correction loop targets program-level correctness, as defined in section 2.3. After the LLM generates a solution, the framework first evaluates whether the response contains a valid Python code

block. If no valid code block is found, the framework automatically generates a corrective prompt indicating that the previous response lacked a valid code block and explicitly requests a complete solution enclosed in a Python Markdown block.

If a valid block is found, then the framework executes the code in the container, but if the execution fails, the framework will construct a different corrective prompt that includes:

- the original task prompt and data context
- the previously generated code
- the error message produced by the container

The LLM receives this corrective prompt and produces a new solution. This process is repeated until the code either executes successfully or the maximum number of attempts is reached.

3.6.2 Feedback mechanism - feedback loop

The feedback loop targets the pipeline-level output from section 2.3 after successful container execution. The framework uses a separate feedback prompt (loaded from a .txt file, similar to the task prompt) to generate structured feedback on the produced solution and suggest improvements. The framework then constructs a new prompt comprising the original task prompt, the previous solution, the previous output, and the structured feedback. The next generation step then attempts to improve the solution based on the feedback.

To avoid degrading results, the feedback loop currently includes a built-in stop condition: if the feedback response indicates that the previous output is better, the feedback loop stops and returns the earlier solution.

3.6.3 Feedback mechanism - controllability

The error-correction and feedback loops can both be enabled or disabled depending on the parameters. This study examines the effects of individual looping mechanisms during the ablation study on the final LLM-DSF framework. The depth limits of the loops can also be controlled, but for consistency, have always been set to 3 in this framework's research to balance computational costs and improve results.

3.7 Tests

The LLM-DSF framework has been designed to evolve during the project, necessitating consistent changes to its codebase. To prevent changes from silently breaking the functionality of the core systems, automated testing has been implemented.

Unit testing. The repository of the LLM-DSF framework includes unit tests for the core modules: input parsing, response parsing, container execution, and the LLM interface. The test suite has been created using PyTest.

Local development checks. The LLM-DSF framework uses pre-commit hooks to run formatting checks, prevent large-file commits, validate YAML, normalize whitespace, and execute the test suite before committing changes. This design reduces the probability of code-breaking changes reaching the repository.

4 EXPERIMENTS AND RESULTS

All experiments and development are performed on a local Linux-based desktop; for full specifications, see Appendix A. Each experiment will begin with the evaluation approach for that setup, and then proceed to the results.

4.1 EXPERIMENTAL SETUP

LLM-DSF aims to be a versatile framework that can generate end-to-end data science models, and therefore needs to support a wide range of tasks. A single experiment does not capture the full range of tasks and is therefore insufficient. The experiments need to assess different aspects of the LLM-DSF. Benchmarking LLM-driven frameworks is not easy, because widely used public datasets have a high chance of occurring in the training data of the underlying large language model, which measures memorization over generalization [22]. For this reason, the evaluation combines datasets that are possibly seen and those that are most likely unseen.

The first experiment will analyze the local model's temperature parameters and test the framework's capabilities on an exact task using sampled data in section 4.2. The second experiment will conduct an ablation study of the framework's tunable components to empirically assess the impact of providing metadata, using feedback, and using error loops in section 4.3. The third experiment will compare the LLM-DSF framework against the cloud models of Gemini and OpenAI to assess the framework's capabilities in section 4.4. The final experiment will pit the framework against AutoML frameworks on tasks from the AutoML Benchmark in section 4.5 [11].

The goal is to provide a broad comparison of the LLM-DSF framework's capabilities and to conduct experiments on both seen and unseen data. Within each experiment, all data used will be explained, and how they contribute to the experiment.

4.2 EXPERIMENT 1: TEMPERATURE ANALYSIS

This experiment measures how sampling temperature affects the correctness of an exact task, using a sampled sales dataset from Kaggle released after Deepseek-R1 [16]. The experiment focuses on performing exact calculations across three columns of the sales dataset, which is not too difficult for a human, but it tests the model's instructions on an exact task while tuning the temperature parameters.

I compare two local .gguf models: DeepSeek-R1-Distilled-Llama-8B and DeepSeek-R1-Qwen-Llama-7B. For each model, I test the temperatures from 0.0 to 1.0 in increments of 0.1. For each model and temperature combination, I run 30 independent generations, yielding 2 X 11 X 30 = 660 runs in total. A run is labeled correct if the executed code returns the exact answers; all other outcomes, including runtime errors and missing or invalid code, are labeled incorrect. I report the accuracy as the percentage of correct runs out of 30 for each model and temperature.

In Appendix D, the sample code of one generation can be found, along with the rounded results.

Figure 2 shows that the Qwen variant is more consistent over most temperatures. Over the 11 temperature settings, Qwen attains higher accuracy in 5 cases, ties in 4 cases, and is lower in 2 cases. Qwen achieves 100% accuracy at temperatures 0.0-0.2, whereas the Llama variant drops below 0.1 and remains below 100 thereafter. Higher temperatures exhibit greater variability in accuracy, whereas the lowest accuracy for Qwen is 86.7% and for Llama 80%.

Based on these results, I use the Qwen variant for the remainder of the experiments and set the default temperature to 0.2.[3] The choice is motivated by the accuracy, while maintaining some randomness in the generated responses.

4.3 EXPERIMENT 2: LLM-DSF ABLATION STUDY

The ablation study will assess the effect of the tunable parameters (i) 'Metadata': include training metadata in the prompt, (ii) 'Errors': enable the error retry loop that retries after runtime failures, and (iii) 'Feedback': enable the feedback loop to improve on existing code on a KNMI dataset from the Netherlands.

[3] DeepSeek-R1-Qwen-32B is the used model in further experiments because a hardware upgrade during the study.

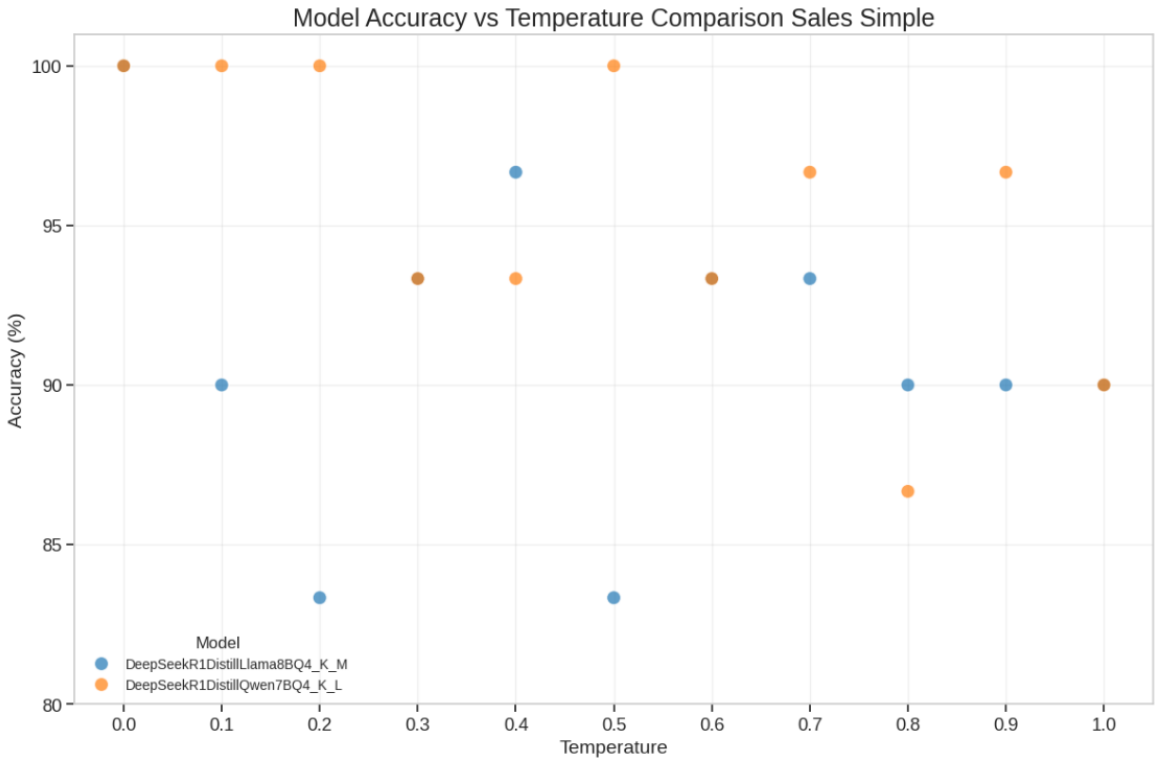

Figure 2: Results of the two models tuned over the temperature. 30 correct = 100% accuracy.

Name	Metadata	Errors	Feedback	Best MAE	AVG MAE (10)
Config 0 (Full framework)	Y	Y	Y	18.592	20.864 (1 DNF)
Config 1	Y	Y	N	20.346	21.406
Config 2	Y	N	Y	20.069	20.598 (4 DNF)
Config 3	Y	N	N	20.373	20.646 (1 DNF)
Config 4	N	Y	Y	18.760	23.612 (3 DNF)
Config 5	N	Y	N	20.060	22.033 (2 DNF)
Config 6	N	N	Y	DNF	NA (10 DNF)
Config 7 (Baseline)	N	N	N	20.529	20.529 (9 DNF)

Table 2: KNMI dataset ablation study results LLM-DSF 10 runs per configuration.
Y = included, N = not included, DNF = code errored.

The data contain temperatures in ℃ X 10, so, for example, 8℃ is 80 in the dataset. The training data were collected from 2001 to 2018, and the test data from 2019.

I run a full design (8 configurations), where each configuration is executed 10 times. I evaluate two outcomes:

- Run success (binary): whether the run produces runnable code that finishes and results in a test MAE value.
- Model quality (continuous): the test MAE reported by the run (lower is better), conditional on success.

Because MAE is undefined for failed runs, I analyze robustness (success) and quality (MAE) separately. I use Fisher's exact tests to assess the main effects on run success, and a factorial ANOVA on MAE conditional on success to analyze differences in quality.

Factor	Fail (0)	Succ (0)	Fail (1)	Succ (1)	Odds Ratio	p-value
Metadata	24	16	6	34	8.500	0.000
Errors	24	16	6	34	8.500	0.000
Feedback	12	28	18	22	0.524	0.248

Table 3: Fisher's exact tests for main effects on run success (KNMI ablation study). Columns *Fail (0)* and *Succ (0)* denote the number of failed and successful runs when the factor is disabled (0), and *Fail (1)* and *Succ (1)* denote the counts when the factor is enabled (1). Odds ratios compare enabled vs. disabled (values > 1 indicate higher odds of success when enabled).

Term	Sum Sq	df	F	p-value	Partial η^2
Metadata	19.796	1	1.692	0.200	0.038
Errors	4.416	1	0.378	0.542	0.009
Feedback	0.000	1	0.000	1.000	0.000
Metadata $\times$ Errors	2.235	1	0.191	0.664	0.004
Metadata $\times$ Feedback	4.202	1	0.359	0.552	0.008
Errors $\times$ Feedback	5.318	1	0.455	0.504	0.010
Metadata $\times$ Errors $\times$ Feedback	5.381	1	0.460	0.501	0.011

Table 4: Factorial ANOVA results for MAE on the KNMI ablation study *conditional on success* (i.e., only successful runs with valid MAE are included; failed runs/DNFs are excluded). Partial η^2 is reported as an effect size estimate.

Table 2 shows that the baseline configuration (config 7) is not robust, with 9 out of 10 runs failing. In contrast, the full framework (config 0) fails only once and achieves the lowest observed MAE of 18.592. Overall, the largest differences across configurations are observed in execution success rather than in MAE values. Note that almost every run had a few high MAE values $>$25, which greatly affected the average MAE.

Table 3 supports the robustness interpretation. Enabling metadata increases the success rate from 40% (16/40) to 85% (34/40), and enabling error retries yields the same improvement by accident. Both factors show strong statistical support ($p < 0.05$). In contrast, the feedback loop does not show a clear improvement and even shows a small decrease in success from 70% to 55%.

Finally, table 4 shows no evidence that any component systematically affects the MAE among successful runs with all $p > 0.05$. This result should be interpreted with caution, as several runs have few successful iterations, thereby reducing the method's statistical power. The high variance of the MAE also makes it harder to establish significant increases, given the relatively small sample size. Empirically, feedback and metadata improve quality, whereas error retries improve robustness, but this cannot be proven here and warrants further investigation.

4.4 Experiment 3: LLM-DSF vs LLMs

This experiment compares LLM-DSF against cloud LLMs on two different regression tasks. The goal is two-fold: (i) evaluate whether LLM-DSF improves robustness, and (ii) compare whether model quality is competitive when evaluated on the same held-out test split.

Datasets. I use two different datasets for this experiment:

- The KNMI temperature dataset (same as in section 4.2). Model quality is measured on MAE on the held-out test dataset; lower is better.
- The DANS housing price dataset [23]. This dataset is accessed through DANS, and private access has been granted for this study and is thus treated as a private dataset for the LLMs. Using a private dataset strengthens the evaluation by reducing the risk that performance is due to memorization of the pretraining data. It improves the credibility of generalization if the framework generalizes to unseen data. The original table has 46.658 rows and 872 columns; however, due to computational constraints, I filtered the columns to 185, selecting those empirically identified as related to pricing by retaining sections related to the building (costs, attributes, area) but excluding sections about the current inhabitants. Model quality is measured by RMSE on the held-out test dataset; lower values are better.

Train and test separation. For both datasets, the framework creates a train and test split before any model interaction, and metadata is generated from the training split only as seen in section 3.2.2. No test rows are added to prevent data leakage. The resulting code will be manually run to confirm the final scores on the real test dataset.

Comparing model conditions. I compare the models using two different modes:

- Direct LLM usage (no framework): the model is prompted to generate a complete Python solution. The generated code is executed by hand to obtain the final metric. If the code does not run or does not report the specific metric, the run is marked as a DNF (did not finish).
- LLM-DSF (full framework): the same task is executed through the framework. The error and feedback loops had a maximum of 3 attempts. Again, the result is verified by hand, and the same rules apply.

I use the same task prompt and training dataset to prevent these from influencing the results in unexpected ways. All models are run three times with similar settings, task prompt, and dataset splits.

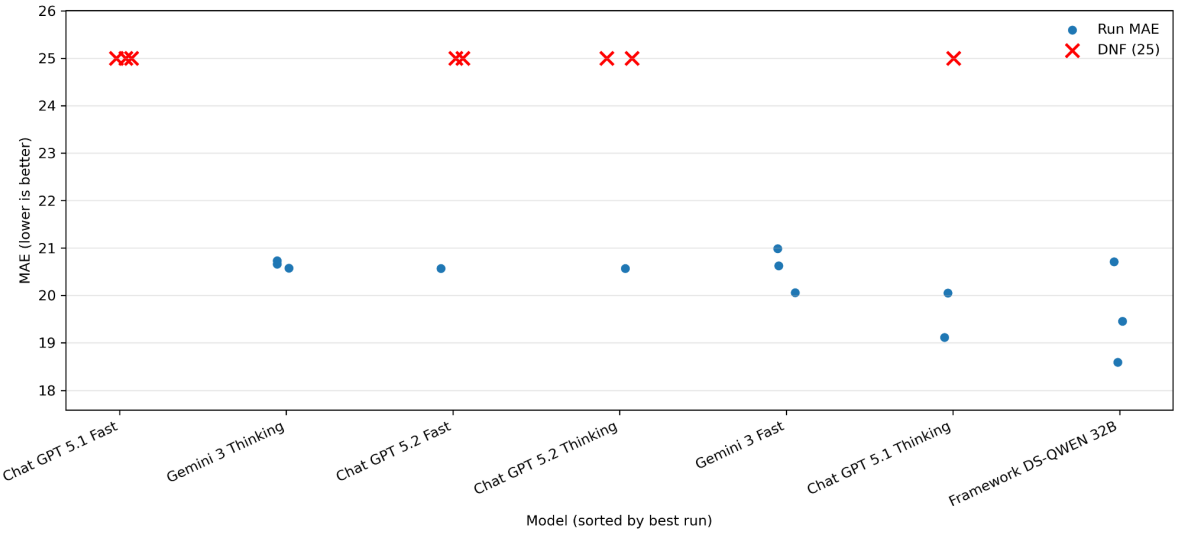

Figure 3: KNMI comparison across LLMs and LLM-DSF. The blue dots represent successful runs; the lower the number, the better. Red crosses indicate DNF's plotted at MAE = 25 for visibility in the figure. Models are ordered by their best run.

Figure 3 shows that robustness differs strongly between conditions. Some cloud models frequently fail, whereas the LLM-DSF configuration using the local Qwen model completes all runs successfully and yields lower MAE values. In terms of the lowest MAE quality, the framework also achieves the lowest value.

The results match the ablation study findings from section 4.3. The KNMI task is sensitive to missing context and runtime failures, and the full framework is needed to obtain the most reliable solutions. LLM-DSF appears to increase the probability of achieving a valid score and slightly affects the final score.

Figure 4 shows a similar pattern on the private DANS dataset. Direct chat models often fail, either completely or partially, whereas the framework yields valid runs more often. In this scenario, some cloud models are integrated into the framework to obtain additional results. Among the successful runs, framework-backed runs appear more robust and yield slightly better results than those without the framework. Because the dataset is private for all the models, the results provide some evidence that the framework can produce useful models beyond the datasets that are more likely to appear in the training data of the models.

The comparisons for both datasets are based on a small number of runs, so the results should not be taken for granted; they support the conclusion thus far that LLM-DSF improves robustness and achieves good quality, but they do not establish statistical significance and warrant further investigation.

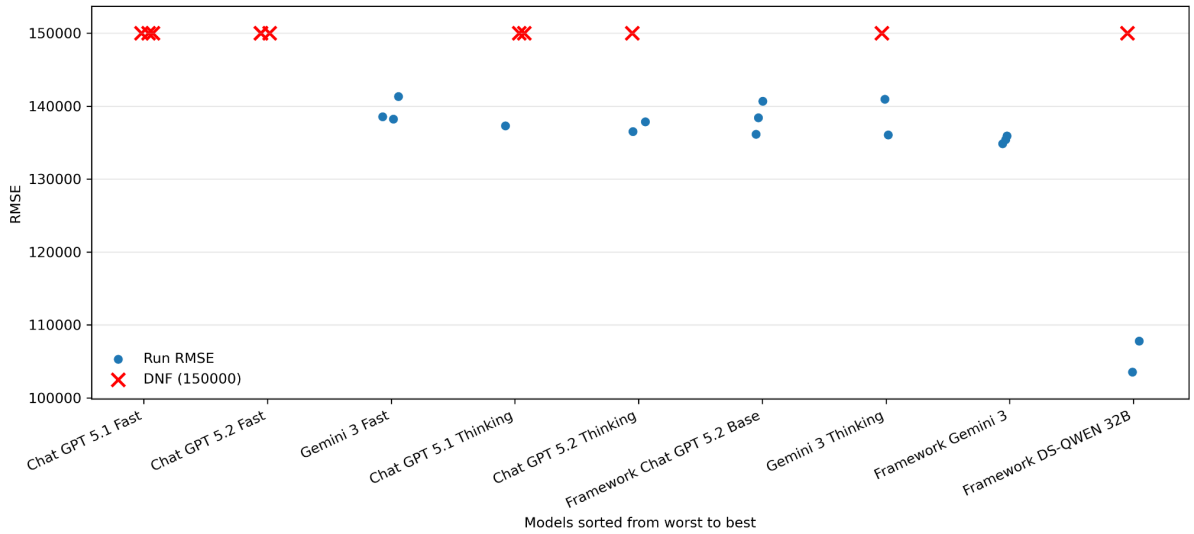

Figure 4: DANS comparison across LLMs and LLM-DSF. The blue dots represent successful runs; the lower the number, the better. Red crosses indicate DNF's plotted at RMSE = 150.000 for visibility in the figure. Models are ordered by their best run.

4.5 Experiment 4: LLM-DSF vs AutoML Benchmark

The final experiment compares LLM-DSF to AutoML frameworks using tasks from the AutoML benchmark [11]. The benchmark provides OpenML tasks and predefined data splits, which allow a controlled comparison across systems that share the same evaluation method.

Manual evaluation. The AutoML benchmark is created to automatically integrate with developed frameworks to test the results. However, LLM-DSF requires task prompts to specify the objective, constraints, and output. The benchmarking suite expects fully automated API calls and does not allow task prompts to be supplied. Therefore, LLM-DSF will be evaluated manually per task, while keeping data splits, metrics, and output measurements identical to those in the benchmark setup.

Tasks. I evaluate a total of 6 benchmark tasks out of the 61 in the AutoML Benchmark, covering three supervised methods: binary classification, multiclass classification, and regression. Due to computational constraints, I evaluate 2 different tasks per method to keep the evaluation manageable.

The following tasks are selected:

- task-168757 credit-g dataset: [15]
- task-189354 airlines dataset:
- task-168784 steel-plates dataset: [5]
- task-359963 segment dataset: [19]
- task-359933 space_ga dataset: [18]
- task-359935 wine quality dataset: [7]

Budget and interpretation. AutoML frameworks in the benchmark optimize pipelines continuously up to a fixed time budget (1 hour), which includes model selection, parameter search, and ensembling. In contrast, LLM-DSF generates and executes a single end-to-end run. The framework will use up to three feedback and error looping attempts, but it does not compare against a time budget, such as the AutoML search. This experiment evaluates whether LLM-DSF can produce reasonable models in a single-generation setting, rather than whether it can outperform budgeted AutoML systems.

Table 5 shows that LLM-DSF does not quite match the performance of the dedicated AutoML systems on the benchmark tasks. The results are behind on 4 out of 6 tasks, and are somewhat close on the regression tasks. This outcome is expected given the differences between the AutoML frameworks that

	Binary classification		Multiclass classification		Regression	
	168757 credit-g	189354 airlines	168784 steel-plates	359963 segment	359933 space_ga	359935 wine-quality
Model	AUC ↑	AUC ↑	logloss ↓	logloss ↓	RMSE ↓	RMSE ↓
AutoGluon(B)	0.796(0.041)	0.732(0.002)	0.464(0.042)	0.052(0.024)	0.095(0.014)	0.570(0.023)
AutoGluon(HQ)	0.788(0.044)	0.732(0.002)	0.465(0.050)	0.056(0.028)	0.100(0.016)	0.570(0.024)
AutoGluon(HQIL)	0.763(0.053)	0.732(0.002)	0.507(0.037)	0.075(0.049)	0.110(0.025)	0.600(0.023)
autosklearn	0.778(0.030)	0.726(0.003)	0.516(0.032)	0.078(0.042)	0.097(0.013)	0.610(0.019)
autosklearn2	0.796(0.041)	0.727(0.002)	0.457(0.027)	0.059(0.023)	–	–
FLAML	0.788(0.039)	0.731(0.002)	0.482(0.037)	0.067(0.031)	0.100(0.016)	0.570(0.020)
GAMA(B)	0.794(0.033)	0.717(0.007)	0.491(0.029)	0.067(0.025)	0.097(0.019)	0.570(0.021)
H2OAutoML	0.779(0.048)	0.731(0.002)	0.490(0.036)	0.061(0.025)	0.097(0.016)	0.570(0.022)
LLM-DSF	0.752(0.033)	0.625(0.003)	0.953(0.135)	0.190(0.047)	0.110(0.016)	0.610(0.020)

Table 5: Cross-task comparison of all evaluated models on the AutoML benchmark with a 1-hour time budget. Rows denote models and columns denote benchmark tasks, identified by task ID, task name, and evaluation metric. Each cell reports mean(std). Metrics differ across task groups: AUC is used for binary classification and higher values are better, while logloss and RMSE are used for multiclass classification and regression, respectively, and lower values are better. Missing entries (–) indicate that no result was reported for that model-task combination.

use the full time budget and are created for this automation, whereas LLM-DSF produces one pipeline per run.

The main advantage of LLM-DSF is not its lack of performance peaking but its versatility. It can generate end-to-end code. It can automate upstream tasks that are out of scope for AutoML frameworks, such as data inspection and transformation. In that sense, the results position LLM-DSF as a rapid automation tool rather than a replacement for AutoML systems when predictive performance is the primary objective.

5 Discussion

This thesis investigates whether LLM-DSF can use large language models to automate larger portions of the end-to-end data science cycle. The experiments indicate that the bottleneck is not primarily whether a model can generate a complete pipeline once, but whether it can do so reliably across repeated runs and different datasets. This means that the main contribution of LLM-DSF is not single run success, but improving reliability to the end-to-end generation. This pattern is evident in experiments 2 and 3, where the framework primarily improves the number of valid completed runs rather than consistently producing the strongest predictive scores.

A first observation is that generation settings can affect functional correctness on tasks. This is expected, but it can significantly affect the framework's performance. Increasing temperature leads to greater randomness and higher variability, whereas decreasing it compromises creativity and the ability to extend to new datasets. This implies that settings such as temperature not only affect generation but also entail a practical trade-off between stability and flexibility in LLM-DSF. This follows from experiment 1, Figure 2.

A second observation is that LLM-DSF improves robustness primarily through elements outside the model rather than by improving the model itself. Adding metadata helps the model reason about column types and ranges, reduce schema errors, and thus reduce the likelihood of generating code that cannot execute. The error-retry loop further increases robustness by providing the model with the generated runtime errors. In contrast, feedback loops can be more ambiguous, either improving the best solution or increasing the model's chances of failing code generation. This supports the view that the practical value of LLM-DSF lies primarily in the framework design surrounding the model, rather than in the model itself. This interpretation bases on the results of experiment 2, Table 2.

A third observation is that LLM-DSF performs well on the private DANS dataset, compared to direct cloud models. Generating a schema improves the framework's robustness, whereas cloud models must read everything from the data directly. This indicates LLM-DSF's usefulness in settings where the structure must be made explicit before model interaction, especially in private datasets as can be seen in Figure 4.

The last observation is that the comparison between LLM-DSF and the AutoML frameworks highlights a trade-off between optimization and generality. AutoML systems are specialized frameworks designed to optimize models, whereas LLM-DSF produces end-to-end solutions in a single run but does not exploit the expensive hyperparameter or model search.

These results suggest that LLM-DSF is best understood as a framework that automates workflows and executes them, rather than as a direct replacement for AutoML systems. In practice, the framework can serve as a fast starting point for completing baselines, after which AutoML methods can take over.

This thesis also has limitations. The experimental evaluation covers multiple task types, but the number of datasets and repetitions per setting is limited by computational constraints, which limits the strength of the claims. Evaluation focuses on tabular tasks and does not automatically generalize to other data modalities. Experiments were evaluated primarily manually, which was time-consuming and further limited the number of runs. The framework also relies on prompt quality and large language model selection, which were outside the scope of this thesis but can substantially influence the results. Overall, LLM-DSF shows clear tendencies, but does not demonstrate the generalization across all end-to-end data science settings.

6 Conclusion

This thesis proposes and evaluates LLM-DSF, a framework that automates end-to-end data science workflows by combining language model code generation with deterministic preprocessing, structured prompt building, containerized execution, and a feedback mechanism. The experiments show that the main contribution of LLM-DSF is not only the ability to generate full end-to-end code, but to do so more reliably than directly prompting LLM.

Sub-question 1: design principles and mechanisms. The first sub-question asks what core design principles and mechanisms support an LLM framework in autonomously generating code for different stages of the data science lifecycle. The literature shows that LLM-based data science systems need more than code generation alone. Prior work demonstrates that large language models can support data science tasks, but also shows that autonomous execution remains challenging due to hallucinations, inconsistent outputs, and limited reliability. LLM-DSF is designed as a modular framework rather than as a single-step process. It combines preprocessing, metadata generation, structured prompt construction, isolated execution, and iterative corrections. The experiments show that these mechanisms are required for robust automation, especially the metadata generation and execution retry loops. The core design principles and mechanisms are therefore modularity, explicit data representation, controlled execution, and iterative corrections rather than one-shot generation.

Sub-question 2: defining high quality. The second sub-question asks what standards should define high quality in the context of code and models generated by an LLM framework. Quality is a common term, but difficult to define within a framework such as LLM-DSF. Related work suggests that high quality should not be defined by only predictive performance, but also by metrics such as correctness, reliability, and practical usability. In this research, high quality is reflected in executable code, correct handling of the data, robustness across runs, and competitive model quality over the evaluated tasks. High quality in this context should therefore be understood as the production of reliable and usable end-to-end pipelines, with predictive performance as one criterion rather than the only one.

Sub-question 3: safety and assessment mechanisms. The third sub-question asks how the framework can incorporate safety and assessment mechanisms to consistently evaluate and adapt its outputs to meet the high-quality standards across the data science lifecycle. Related work identifies safety and assessment mechanisms as central challenges for LLM-based automation, particularly because generated code may be incorrect, unsafe, or difficult to verify. LLM-DSF addresses this by combining containerized execution, train-test separation during preprocessing, in which metadata is generated only on the training split, and iterative error and feedback mechanisms. The experiments show that error iteration and metadata are the most effective assessment mechanisms for consistently improving robustness. The containerized setup, on the other hand, provides a controlled environment for running the generated code. This indicates that safety and assessment mechanisms should rely on explicit and verifiable feedback rather than on the language model alone.

Answer to the main research question. LLM-DSF shows that a practical approach to autonomously generating high-quality models requires treating the LLM as a single component within a controlled loop, rather than as a single solution. By enforcing a structured interface, executing code in isolated containers, and applying error correction, the framework increases the probability of producing complete and usable solutions. The experiments show that these mechanisms are important for robustness, but also reveal a trade-off between end-to-end automation and predictive optimization achieved by AutoML systems. LLM-DSF answers the research question not by replacing existing optimization frameworks, but by demonstrating how LLMs can be embedded into a practical workflow that automates large parts of the data science cycle in a controlled manner. The quantitative results support this interpretation. In the KNMI ablation study, the full framework produced valid runs in 9/10 repetitions, compared to 1/10 from the baseline configuration without metadata, error retries, or feedback. For both enabling metadata and error retries, the total number of successes increased from 16/40 to 34/40. In the AutoML benchmark comparison, LLM-DSF did not match the best-performing AutoML variant in predictive performance across experiments, but it remains within a 20% relative difference on 4/6 tasks.

Future work. There are multiple extensions that follow directly from my observations. First, LLM-DSF would benefit from integrating optimization loops similar to AutoML behavior. The framework could, for example, preprocess and clean the data, whereas the AutoML frameworks can be used for model selection and optimization. Another direction for future work would be to split LLM-DSF into smaller, specialized portions that hand off completed steps and results in a chain. This can enhance the overall

stability by breaking the tasks into smaller, easier tasks. A third direction that would be interesting is to have LLM-DSF preprocess unstructured data into tabular form, a time-consuming and difficult task that would benefit many processes. Finally, broader validation across datasets, model backends, prompt quality, and runs would strengthen the evidence for the framework's effectiveness.

REFERENCES

[1] Mitra Baratchi, Can Wang, Steffen Limmer, Jan N. van Rijn, Holger H. Hoos, Thomas Bäck, and Markus Olhofer. Automated machine learning: past, present and future. *Artif. Intell. Rev.*, 57(5):122, 2024. doi: 10.1007/S10462-024-10726-1. URL https://doi.org/10.1007/s10462-024-10726-1.

[2] Tijl De Bie, Luc De Raedt, José Hernández-Orallo, Holger H. Hoos, Padhraic Smyth, and Christopher K. I. Williams. Automating data science. *Commun. ACM*, 65(3):76–87, 2022. doi: 10.1145/3495256. URL https://doi.org/10.1145/3495256.

[3] Pavel Brazdil, Jan N. van Rijn, Carlos Soares, and Joaquin Vanschoren. *Metalearning: Applications to Automated Machine Learning and Data Mining*. Springer, 2022. ISBN 978-3-030-67024-5. doi: 10.1007/978-3-030-67024-5. URL https://doi.org/10.1007/978-3-030-67024-5.

[4] Bureau of Labor Statistics, U.S. Department of Labor. Data Scientists. Occupational Outlook Handbook, 2025. URL https://www.bls.gov/ooh/math/data-scientists.htm.

[5] M Buscema, S Terzi, and W Tastle. Steel Plates Faults. UCI Machine Learning Repository, 2010. DOI: https://doi.org/10.24432/C5J88N.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

[7] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decis. Support Syst.*, 47(4):547–553, 2009. doi: 10.1016/J.DSS.2009.05.016. URL https://doi.org/10.1016/j.dss.2009.05.016.

[8] Matthias Feurer, Jan N. van Rijn, Arlind Kadra, Pieter Gijsbers, Neeratyoy Mallik, Sahithya Ravi, Andreas Müller, Joaquin Vanschoren, and Frank Hutter. OpenML-Python: an extensible Python API for OpenML. *J. Mach. Learn. Res.*, 22:100:1–100:5, 2021. URL https://jmlr.org/papers/v22/19-920.html.

[9] Matthias Feurer, Katharina Eggensperger, Stefan Falkner, Marius Lindauer, and Frank Hutter. Auto-Sklearn 2.0: Hands-free AutoML via Meta-Learning. *J. Mach. Learn. Res.*, 23:261:1–261:61, 2022. URL https://jmlr.org/papers/v23/21-0992.html.

[10] Tim Furche, Georg Gottlob, Leonid Libkin, Giorgio Orsi, and Norman W. Paton. Data Wrangling for Big Data: Challenges and Opportunities. In *Proceedings of the 19th International Conference on Extending Database Technology (EDBT 2016)*, pp. 473–478. OpenProceedings.org, 2016. doi: 10.5441/002/edbt.2016.44. URL https://doi.org/10.5441/002/edbt.2016.44.

[11] Pieter Gijsbers, Marcos L. P. Bueno, Stefan Coors, Erin LeDell, Sébastien Poirier, Janek Thomas, Bernd Bischl, and Joaquin Vanschoren. AMLB: an automl benchmark. *J. Mach. Learn. Res.*, 25:101:1–101:65, 2024. URL https://jmlr.org/papers/v25/22-0493.html.

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang

Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, Tao Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nat.*, 645(8081):633–638, 2025. doi: 10.1038/S41586-025-09422-Z. URL https://doi.org/10.1038/s41586-025-09422-z.

[13] Siyuan Guo, Cheng Deng, Ying Wen, Hechang Chen, Yi Chang, and Jun Wang. DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*, volume 235 of *Proceedings of Machine Learning Research*, pp. 16813–16848. PMLR / OpenReview.net, 2024. URL https://proceedings.mlr.press/v235/guo24b.html.

[14] Lydia Harriss, Josh Fearns, and Clare Lolly. Data science skills in the UK workforce. POSTnote 697, UK Parliament, Parliamentary Office of Science and Technology (POST), 2023. URL https://doi.org/10.58248/PN697. Published June 28, 2023.

[15] Hans Hofmann. German Credit Data. UCI Machine Learning Repository, 1994. DOI: https://doi.org/10.24432/C5NC77.

[16] V Kannaece. sales dataset. Kaggle, 2025. URL https://www.kaggle.com/datasets/vinothkannaece/sales-dataset?resource=download.

[17] Lei Liu, So Hasegawa, Shailaja Keyur Sampat, Maria Xenochristou, Wei-Peng Chen, Takashi Kato, Taisei Kakibuchi, and Tatsuya Asai. AutoDW: Automatic Data Wrangling Leveraging Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 2041–2052, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695267. URL https://doi.org/10.1145/3691620.3695267.

[18] R. Kelley Pace and Ronald Barry. Quick Computation of Spatial Autoregressive Estimators. *Geographical Analysis*, 29(3):232–247, 1997. doi: 10.1111/j.1538-4632.1997.tb00959.x. URL https://doi.org/10.1111/j.1538-4632.1997.tb00959.x.

[19] UCI Machine Learning Repository. Image Segmentation. UCI Machine Learning Repository, 1990. DOI: https://doi.org/10.24432/C5GP4N.

[20] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nat.*, 625(7995):468–475, 2024. doi: 10.1038/S41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

[21] Yuan Sui, Mengyu Zhou, Mingjie Zhou, Shi Han, and Dongmei Zhang. Table Meets LLM: Can Large Language Models Understand Structured Table Data? A Benchmark and Empirical Study. In *Proceedings of the 17th ACM International Conference on Web Search and Data Mining, WSDM 2024, Merida, Mexico, March 4-8, 2024*, pp. 645–654. ACM, 2024. doi: 10.1145/3616855.3635752. URL https://doi.org/10.1145/3616855.3635752.

[22] Alexander Tornede, Difan Deng, Theresa Eimer, Joseph Giovanelli, Aditya Mohan, Tim Ruhkopf, Sarah Segel, Daphne Theodorakopoulos, Tanja Tornede, Henning Wachsmuth, and Marius Lindauer. AutoML in the Age of Large Language Models: Current Challenges, Future Opportunities and Risks. *Trans. Mach. Learn. Res.*, 2024, 2024. URL https://openreview.net/forum?id=cAthubStyG.

[23] Ministerie van Binnenlandse Zaken en Koninkrijksrelaties (BZK) and Centraal Bureau voor de Statistiek (CBS). Woononderzoek Nederland 2021 - woningmarktmodule- release 1.0, 2022. URL https://doi.org/10.17026/DANS-XAA-MRRA.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need. *CoRR*, abs/1706.03762, 2017. URL http://arxiv.org/abs/1706.03762.

[25] Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. DataSciBench: An LLM Agent Benchmark for Data Science. *CoRR*, abs/2502.13897, 2025. doi: 10.48550/ARXIV.2502.13897. URL https://doi.org/10.48550/arXiv.2502.13897.

A System Hard- and Software

This appendix lists the hardware, operating system, and Python version used during the development and experimentation of the master's thesis.

Hardware

- CPU: Ryzen 9 7950x 16 core CPU
- GPU: RTX 5090 32GB VRAM
- RAM: 64GB DDR5 6400 MHz
- SSD: Samsung 9100 PRO

Software

- OS: Ubuntu 24.04
- Python: 3.12.8

B INPUT PROCESSOR EXAMPLE

This appendix shows how the input processor transforms raw data into parts that are added to the prompt by the framework.

STN	YYYYMMDD	TX
260	19900101	12
260	19900102	22
260	19900103	25
260	19900104	21
260	19900105	45

Table 6: Example of raw data before preprocessing.

```
Dataset Metadata:
Total Rows: 8765
Sample Size: 5 rows
Total columns: 3

Column Information:
- stn:
  Type: int64
  Unique Values: 1
  Range: 260.0 to 260.0
  Average: 260.00

- yyyymmdd:
  Type: int64
  Unique Values: 8766
  Range: 19900101.0 to 20191231.0
  Average: 260.00

- tx:
  Type: int64
  Unique Values: 392
  Range: -81.0 to 372.0
  Average: 146.15
```

Figure 5: Example of metadata being added to the prompt.

```
Data Sample:
stn|yyyymmdd|tx
260|20121203|72
260|20070321|83
260|20070910|177
260|20130913|178
260|19980113|96
```

Figure 6: Example of data sample being added to the prompt.

C CONTAINER CODE EXAMPLES

This appendix shows code samples before and after modification by the container part of the program.

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset (already provided as df)
# Extract features and target
X = df[['sepallengthcm', 'sepalwidthcm', 'petallengthcm', 'petalwidthcm']]
y = df['species']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Build and train the classifier model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Evaluate the model on the test set
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Print the final accuracy on the test data
print(f"Test Accuracy: {accuracy:.4f}")
```

Figure 7: Example of generated code before modification by the container.

```
import pandas as pd

# Load the preprocessed DataFrame
df = pd.read_parquet('/app/data/data.parquet')

# Analysis code
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
# Extract features and target
X = df[['sepallengthcm', 'sepalwidthcm', 'petallengthcm', 'petalwidthcm']]
y = df['species']
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Build and train the classifier model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
# Evaluate the model on the test set
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
# Print the final accuracy on the test data
print(f"Test Accuracy: {accuracy:.4f}")

Starting container execution...
Container finished execution in 2.38 seconds
Code execution completed.
Removing the container...
Container removed successfully.
Execution Output: Test Accuracy: 1.0000
```

Figure 8: Example of generated code after modification by the container.

D TEMPERATURE EXPERIMENT EXAMPLES

This appendix presents one run of the temperature analysis on a sales sample set, with the correct results manually calculated.

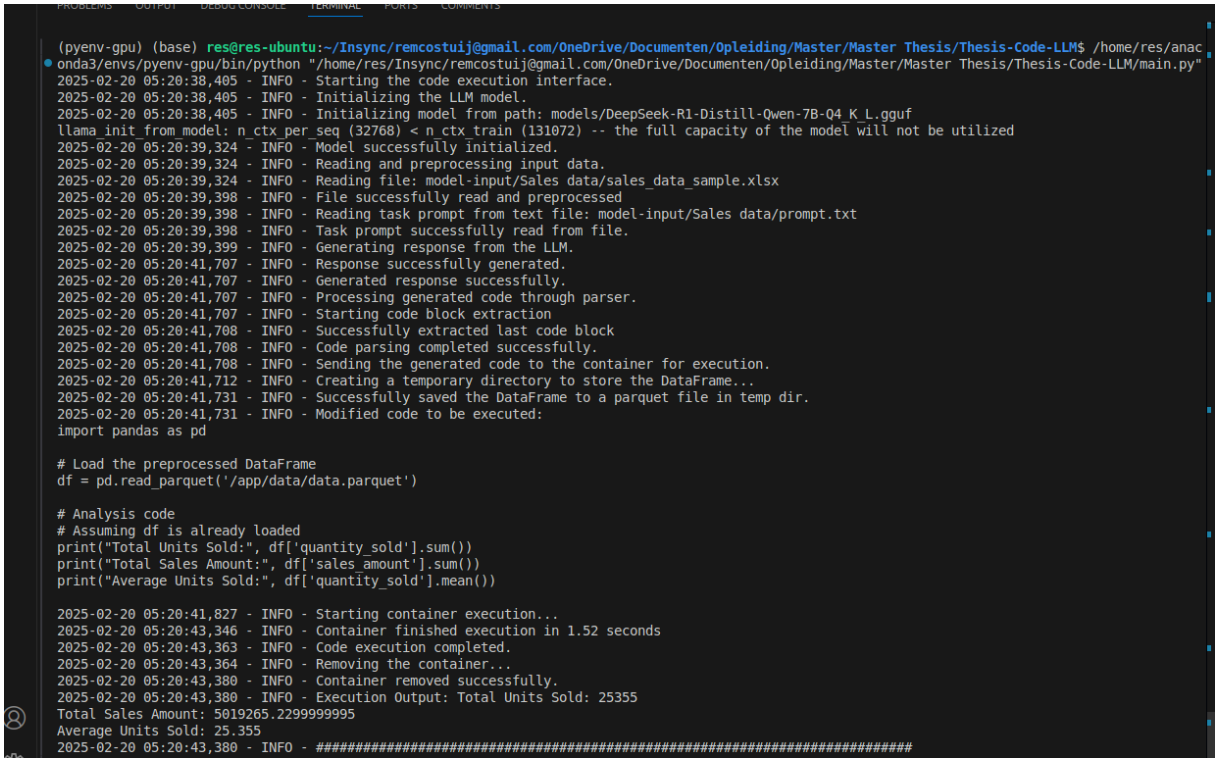

Figure 9: Example of generated code of the temperature experiment with sales sample data.

Method	Column	Rounded result
Sum	Sales_amount	5019265.23
Sum	Quantity_Sold	25355
Average	Quantity_Sold	25.36

Table 7: Rounded results on two of the temperature experiments with sales sample data calculated.