Received 13 February 2025, accepted 28 March 2025, date of publication 1 April 2025, date of current version 18 April 2025.

*Digital Object Identifier 10.1109/ACCESS.2025.3556973*

RESEARCH ARTICLE

# Exploring the Potential of Offline LLMs in Data Science: A Study on Code Generation for Data Analysis

**ANASTASIOS NIKOLAKOPOULOS**[1], **ANTONIOS LITKE**[1], **ALEXANDROS PSYCHAS**[1], **ELENI VERONI**[2], **AND THEODORA VARVARIGOU**[1]

[1]School of Electrical and Computer Engineering, Institute of Communication and Computer Systems, National Technical University of Athens, 157 73 Zografou, Greece
[2]Research and Innovation Development Department, Netcompany-Intrasoft S.A., 15125 Luxembourg, Luxembourg

Corresponding author: Anastasios Nikolakopoulos (tasosnikolakop@mail.ntua.gr)

This work was supported in part by the European Commission through the Horizon Europe Program through the Project ''Reliable biomeTric tEchNologies to asSist Police authorities in cOmbating terrorism and oRganized crime (TENSOR)'' under Grant 101073920.

**ABSTRACT** Large Language Models (LLMs) have recently attracted considerable attention from the scientific community, due to their advanced capabilities and potential to serve as vital tools across various industries and academic fields. An important implementation domain for LLMs is Data Science, in which they could enhance the efficiency of Data Analysis and Profiling tasks. With the utilization of LLMs in Data Analytics tools, end-users could directly issue data analysis queries in natural language, bypassing the need for specialized user interfaces. However, due to the sensitive nature of certain data in some organizations, it is unwise to consider using established, cloud-based LLMs. This article explores the feasibility and effectiveness of a standalone, offline LLM in generating code for performing data analytics, given a set of natural language queries. A methodology tailored to a code-specific LLM is presented, evaluating its performance in generating Python Spark code and successfully producing the desired result. The model is assessed on its efficiency and ability to handle natural language queries of varying complexity, exploring the potential for wider adoption of offline LLMs in future data analysis frameworks and software solutions.

**INDEX TERMS** Code generation, data analysis, data profiling, data science, large language models.

## I. INTRODUCTION

### A. APPLICATIONS AND IMPACT OF LLMS

The meteoric rise of Large Language Models (LLMs) has opened up new possibilities for research and development of practical applications across various fields. The capabilities of such models have generated increased interest in the global research community, leading to investments for further innovations in the field of LLM-based applications. Famous LLMs, including OpenAI's GPT-4 [1] and Google's Gemini [2], have achieved the ability to understand, produce, and manipulate human language to such a degree that they — or other LLMs — are now applied across several domains, from health care to engineering. These capabilities have paved the way to new standards in both research and industrial applications, with one of the main goals being the improved accuracy of the models' outputs [3], [4].

Of course, the applications of LLMs go far beyond traditional natural language processing. In scientific research, for example, LLMs are used to speed up discoveries in medicine, biology, materials science, and forensics. For instance, their application in drug discovery could help enable speed up the development of new therapeutic agents, by predicting molecular properties and simulating molecular interactions. In addition, the use of LLMs in computational chemistry is aimed at achieving better chemical processes description, whilst in materials science LLMs are being utilized in order to design new materials with tailored properties [5]. Moreover, the integration of LLMs in forensic science could enhance modern investigations and assist

The associate editor coordinating the review of this manuscript and approving it for publication was Essam A. Rashed.

law enforcement agencies in suspect identification [6]. These are only few examples that demonstrate the reaching potential of LLMs in scientific research and industrial innovation, which represents a potential quantum leap toward much more intelligent, efficient problem-solving across all fields [7].

Furthermore, Large Language Models have proven to have great potential in providing enhancements for several aspects of Data Science. For example, OpenAI's GPT-4 can perform tasks like cleaning data, extracting features, and even training a model with limited human intervention [8], [9]. In data analysis, LLMs also provide advantages in automating the process of generating insights from datasets. The models can carry out classic data profiling operations, provide analyses of anomalies and even generate predictive analytics for future use by data engineers, or other industry professionals. So, these models could be used in processing customer feedback, financial reports, and social media data, in order to provide actionable insights. Their ability to understand and interpret several kinds of datasets could make them valuable tools for data scientists [10], [11].

Despite their impressive capabilities and great potential in Data Science (among several other aforementioned domains), Large Language Models face significant challenges when it comes to providing quality analytics on large datasets. One primary limitation is their inability to process entire datasets at once, due to memory and computational constraints. Most LLMs can handle only a limited context window, which can lead to loss of context, incomplete insights, and therefore false results [12]. Additionally, LLMs may struggle with understanding complex data relationships that exist within a dataset, as their training has mainly focused on comprehending natural language, rather than complex data structures [13], [14]. These limitations can compromise the quality and accuracy of the analytics generated by an LLM. Thus, while models like GPT-4 have advanced capabilities in natural language processing, their application to data analytics needs improvement, mainly due to the high complexity of operations related to the field, like in-depth data profiling, analysis, or quality insights extraction [15].

Moreover, the sensitive nature of many datasets further complicates the use of online LLMs — like OpenAI's GPT — for data analysis and profiling. For example, many datasets contain personal information, proprietary business data, or sensitive research findings, making them subject to restrictive regulations — such as the General Data Protection Regulation (GDPR) — and various data privacy laws [16]. The security risks associated with processing such data online pose considerable concerns, including unauthorized access, data breaches, and misuse of information [17]. Furthermore, ethical considerations surrounding the use of sensitive data call for strict compliance with consent and usage guidelines, which can be difficult to enforce when utilizing online LLMs. For that reason, the use of online LLMs for sensitive data profiling and analysis should be avoided.

### B. EXPLORING OFFLINE LLMS IN DATA ANALYSIS

The outlined concerns emphasize the need for secure, on-premise solutions to ensure data integrity and privacy. Organizations handling sensitive data should still be able to benefit from the capabilities of large language models. End-users could benefit greatly from this, by simply querying data in natural language and receiving responses as visualized outputs or direct data samples. For instance, rather than selecting specific columns and applying predetermined rules through a specialized interface, end-users and data scientists could simply write their queries in natural language, and then view the results on their screens as they would normally do. In summary, offline LLMs can truly assist data scientists in data analysis and quality insights extraction, reducing effort and simplifying the querying process.

This leads to the conclusion that enterprises, organizations, and businesses should consider implementing offline LLM solutions. The scope of this article is to explore how offline LLMs can enhance data analysis, by generating code for data analytics operations based on natural language queries. By keeping data on-premise, concerns about data integrity and security are eliminated, since data is not transmitted to online LLM models. Additionally, this approach addresses the limitation of LLMs' inability to properly process entire datasets due to architectural and computational constraints. In the proposed procedure, the LLM does not load entire datasets. Instead, it receives a comprehensive summary. For each data analysis query, the LLM generates specialized code, which is then executed by a software job on the full dataset to return the final results. Consequently, both data integrity and processing limitations are effectively managed.

The proposed on-premises solution includes an offline LLM, along with a data processing platform. Data processed remain within the organization. For each dataset, concise metadata is provided to the LLM, enabling it to gain contextual knowledge of the data. In practice, every time the organization's data scientist makes a query in natural language, the LLM will generate specialized code for data profiling and analysis. The proposed solution will then apply the generated code to the data through a pipeline, ultimately returning the results back to the user. As previously stated, this approach not only maintains data security, but also optimizes the efficiency of data analysis, thus addressing two current limitations of LLMs. Moreover, it helps end-users submit their data analytics preferences effortlessly, since they will be able to do so using natural language.

This paper is structured as follows. In the Related Work (II) section, existing literature and approaches relevant to LLM code generation are reviewed, along with other similar applications and market tools. The Study Methodology and Design (III) section outlines the framework and methodology employed in this study, aiming to evaluate the efficiency of offline LLMs in generating data analysis code. In addition, there is an analysis of the proposed system's architecture, along with an introduction to the datasets selected for evaluation. Technical Components (IV) section discusses the technological infrastructure chosen for this study, analyzing the components that make up the proposed architecture. In the Testing and Results (V) section, the outcomes of the complete testing phase are presented, studying the performance of two offline LLMs in generating accurate and efficient code for various natural language queries. The Critical Assessment (VI) and Conclusion (VII) sections summarize the findings, discuss the potential effects of the results, and suggest directions for future research in this emerging area.

## II. RELATED WORK

Proposals for evaluating the code generation capabilities of LLMs have been made over the past months, as presented in this section. However, the task of generating code specifically for data analysis and profiling using offline models is yet to be extensively covered by the research community. Consequently, direct benchmarking against existing studies is challenging, as no prior work has specifically addressed this approach. In addition to reviewing proposed systems and studies on LLMs and code generation, fully available market solutions are also presented. Moreover, related works where LLMs are applied to data analysis operations are mentioned. Since research around the capabilities of language models is both ongoing and extensive, it is anticipated that additional studies will follow in the future.

### A. CODE GENERATION TECHNIQUES

This subsection inspects existing works aligned with the scope of the current study, primarily in the domain of code generation using LLMs. These studies highlight proposals and approaches in generating functional code, emphasizing the integration of LLMs with additional software techniques, and reflecting the growing potential of these models in diverse coding scenarios.

A research work about the code generation capabilities of ChatGPT was published by Feng et al. [18]. With an emphasis on OpenAI's LLM, the study presents a scalable framework for crowdsourcing social data that assesses the code generation capabilities of large language models. The framework utilizes data from various social media sites. It is used for tasks including academic assignment solving, interview preparation, and pure code debugging. The publication relates to the current work in terms of code generation by language models, but it is not focused on data analytics tasks, or the use of offline LLMs. A study by Gu [19] introduces a code generation approach for compiler testing with the use of a language model, and the goal to increase both the quality and the quantity of the code generated. The technique follows a filter strategy by cleaning the source code, providing a high-quality dataset for model training. This approach targeted the capabilities of encoder-decoder models to produce testing-oriented code, which relates to the current work in terms of utilizing AI tools to explore their code generation capabilities.

An interesting article was authored by Ross et al. [20], introducing a prototype system designed to investigate the effectiveness of conversational interactions between professionals and LLMs, and to assess how software engineers respond to engaging in dialogue with a code-fluent LLM. The results demonstrate that future frameworks with LLM-powered features could become highly assistive tools for software engineers, similar to what this current article aims to assess. At a publication written by Soliman et al. [21], the research team presents hybrid models for code generation through the integrated use of other pre-trained language models. The hybrid models' performance is evaluated using two commonly used datasets [22], [23]. Then, the researchers benchmark them against existing state-of-the-art models. The study aims to explore how is it feasible to enhance the precision and efficiency of LLM code generation, especially in complicated coding scenarios. Although not specifically centered on offline LLMs, this publication is relevant to the current article as it addresses code generation. The present study expands on this by exploring a use case scenario focused on data analytics operations.

A paper published by Pinna et al. [24] explores the use of large language models for automatic code generation, using problem descriptions as input queries. The researchers aimed to address the issue of LLMs generating incorrect code. The results demonstrate an improvement in code quality, highlighting the potential of combining LLMs with other techniques for improving the code generated by the models. Building on this premise, the current study adopts the proposed approach of combining LLMs with other software methodologies. Other notable publications include the proposals of benchmarks for evaluating the code generation process of LLMs: An article by Yu et al. [25] presents a benchmark designed to evaluate code generation models on non-standalone functions that are usually overlooked in the existing benchmarks but make up the majority of the functions found in open-source projects. Another similar research on LLMs for code generation tasks comes from Omari et al. [26], who investigates the capability of a large language model (mainly ChatGPT) in detecting and repairing bugs in simple Python programs.

### B. LITERATURE SURVEYS

This subsection presents key literature surveys that, although not directly aligned with the specific scope of the current study, offer valuable context. These studies explore the role of language models in transforming tasks such as broad-spectrum code generation, debugging, and design, while also highlighting their strengths, limitations, and future potential.

A survey about the potential of large language models in software engineering was published by Fan et al. [27]. It identifies a number of research challenges about the use of LLMs to help software engineers with a range of technical issues, like coding, design, bug fixing, and refactoring, where LLMs bring creative qualities to the table. The report also emphasizes the importance of hybrid approaches in the future, which combine LLMs with traditional software engineering methodologies. A review conducted by Wong et al. [28] goes through the use of Natural Language Processing (NLP) techniques — mainly LLMs trained on big code — in AI-assisted programming operations. It highlights the significant role of LLMs in various applications, such as code generation, completion, translation, refinement, summarization, defect detection, and clone detection. It also mentions examples of AI-enhanced tools, like GitHub Copilot [29] and DeepMind AlphaCode [30].

Another review on large language models and their ability to generate code was authored by Wang and Chen [31]. This study reviews recent research on code generation using large language models, with an emphasis on the evaluation of generated code as well as its use in software engineering activities. According to the study, additional investigation is required to fill in the gaps in the evaluation of LLM-generated code. A similar study was performed by Liu et al. [32], aiming to create a comprehensive literature review on the current developments in deep learning-based code generation. Systematic evaluation was conducted on recent scientific publications. Then, a structured methodology was applied to the review of these papers, in order to provide insights about code generation using large language models, to address existing knowledge gaps, and to guide future research in the field.

### C. OTHER APPLICATIONS AND MARKET TOOLS

This subsection explores LLM fine-tune proposals, as well as applications and market tools that leverage such language models for innovative solutions. These advancements demonstrate the versatility of LLMs in domains such as hardware design, database optimization, personalized content generation, and AI-powered data analytics—aligning most closely with the focus of the current study.

Early attempts to fine-tune existing large language models specifically for code generation have been published. An article by Thakur et al. [33] explores the potential of large language models in automating hardware design through completion of partial Verilog code, a language widely used in digital system design. Another study by Mu et al. [34] presents a framework for improving code generation, by enabling an LLM to identify ambiguous requirements and to request clarifying questions prior to generating the code. Evaluation demonstrates that the model improves the performance of online LLMs (such as GPT-4) in code generation on multiple benchmarks.

Another research study, this time by Rau and Kamps [35], focuses on revisiting and validating the Unsupervised Passage Retrieval (UPR) approach by Sachan et al. [36]. This method relies on the generative capability of large language models to produce zero-shot questions for passage retrieval, emphasizing question generation to re-rank passages for increasing retrieval accuracy. The research not only replicates the effectiveness of UPR on the BEIR benchmark [37], but also extends its evaluation to other benchmarks, the 2019 and 2020 TREC Deep Learning tracks [38].

Another study that examines the potential of language models in generation of useful insights (in this case, reviews) was made by Qu et al. [39], which presents a graph-enhanced prompt learning approach for personalized review generation in e-commerce, utilizing a pre-trained language model (PLM) for higher-quality generation. The approach aims to address the semantic diversity of reviews and enhance the overall quality of generated content. An additional noteworthy publication was made by Zhou et al. [40], proposing a framework that takes advantage of large language models in order to optimize database systems. The framework overcomes challenges related to LLM-based database optimization, such as generating appropriate prompts, capturing both logical and physical characteristics of a database, and ensuring a database's privacy.

Apart from published proposals and studies on the subject, AI-enhanced data profiling and analysis tools have been gradually becoming common in modern data workflows, and they can already become parts of deployment options in leading platforms. OpenAI's Codex [41], a fine-tuned descendant of GPT-3 which is utilized in GitHub Copilot [29], shows that LLMs can assist in writing code for data profiling and analysis tasks, especially in the cloud. Codex works by processing user data on remote servers using powerful cloud computing resources to generate code suggestions, which can boost productivity. Its dependence on the cloud might raise concerns from a privacy point of view: Sensitive data, which might be transmitted to and computed on third-party servers, could lead to data confidentiality issues. Apart from Codex, other GPT-powered tools, mostly functioning in the cloud, provide natural language processing capabilities that aid data analysis tasks. While such features are useful, they require data to be sent to OpenAI's servers for processing, which may pose risks related to data privacy and security.

In contrast, a number of existing data analytics platforms — that provide AI-powered operations — could offer better deployment flexibility for addressing data privacy concerns: DataRobot [42], ThoughtSpot [43], Tableau [44], and Microsoft's Power BI [45], among others, are platforms that provide both cloud and on-premises solutions. For example, DataRobot allows organizations with sensitive data to take advantage of its AI capabilities through an on-premises deployment, so that they can maintain control by keeping the data locally. ThoughtSpot and Tableau also support on-premises deployments, which aims to give more control to users on their data, by enabling processing within their own infrastructure. Similarly, Microsoft Power BI enables a hybrid approach where users can choose either a cloud or an on-premise environment, depending on user privacy and security requirements.

### D. PROMPT ENGINEERING

Particular emphasis should also be placed on Prompt Engineering, as it plays a pivotal role in the current study's proposed system. Prompt engineering is the practice of optimizing the inputs to a generative AI model, so that it responds meaningfully and effectively across a wide range of queries [46], [47]. Through the careful crafting and refinement of prompts, engineers help the models understand not only the language but also the intent and context behind a query. This process is considered a practical way to improve the quality and relevance of responses in applications, ranging from software engineering tasks to consumer chatbots. Effective prompt engineering helps reduce extensive post-processing that might be required in the later phases of the flow, saving time and making AI workflows more effective [48], [49]. Thus, carefully crafted prompts can enable AI systems to provide accurate insights, generate code snippets, or even simulate potential cyberattacks.

Prompting techniques such as zero-shot (giving the model a task without any prior examples, relying solely on the model's pre-trained knowledge), few-shot (providing the model with a few examples or demonstrations to guide its response, improving accuracy when the model encounters similar tasks), and chain-of-thought (breaking down complex tasks into smaller, logical steps that the model can follow, which helps in tasks requiring reasoning or multistep problem-solving) allow for more sophisticated AI interactions, like enabling models to handle tasks they weren't explicitly trained for, or to follow complex reasoning paths. Other techniques and proposals for improving prompt engineering have already been presented, showing that the research community is putting effort on expanding this topic [50], [51], [52]. As generative AI models grow in scope and complexity, prompt engineering will become increasingly vital for taking advantage of their full potential.

In summary, the field of LLMs is continuously evolving, with ongoing research exploring their capabilities on several aspects. Existing studies focus on general-purpose code generation, compiler testing, and hybrid model approaches (among others), demonstrating the evolving landscape of AI-assisted programming. Literature surveys further emphasize LLMs' potential in software engineering. Other applications range from hardware design automation to AI-powered data analytics, while several market tools offer cloud-based and on-premise solutions with AI-assistance. Moreover, prompt engineering techniques seem to enhance LLM capabilities, improving accuracy and efficiency. As previously outlined, since LLM research continues to advance, future studies will likely explore additional approaches.

## III. STUDY METHODOLOGY AND DESIGN

### A. METHODOLOGY

This study has drawn inspiration from a position paper published in 2024 [53]. As previously stated, the ultimate goal would be to develop a scalable data profiling and analysis framework, capable of managing and handling all volumes of data. The end-users would be able to perform analytics queries in natural language, which would then be translated into executable code by an offline large language model. Then, the code would be forwarded to a data management and processing platform for execution. This way, users would have the freedom to seamlessly perform any analytics operation they wish, by simply expressing it with words to an LLM prompt. While the data management and processing platform itself can be safety deployed, the LLM-based approach needs validation. while this study does not incorporate traditional statistical methods, its rigor is maintained through a carefully structured experimental design.

In summary, this study's methodology starts by defining its objective: exploring an offline LLM's ability to generate code for data analysis. Then comes the dataset selection for testing and evaluation, followed by the query design phase, where five distinct queries for each dataset are being authored. Next, the execution plan is established, defining the number of iterations/tests for each query and establishing a communication pipeline between the LLM and the data processing platform. Evaluation metrics are being defined, with explanations provided for their selection. Then comes the results collection and analysis, where the research team assesses the LLM's performance. Last but not least, conclusions are being drawn, and ideas for future improvements are proposed.

In detail, the methodology carried out in this study is as follows:

1) Objective Definition: The objective of this study is to assess the performance and efficiency of an offline LLM in code generation for data analysis operations. The LLMs selected for this study are Mistral AI's Codestral and Alibaba's Qwen 2.5 Coder, and they will be tested in providing Python Spark (PySpark) code applying data profiling and analytics filters in datasets. The Codestral, Qwen and PySpark selections are analyzed in subsection IV-A.
2) Datasets Selection: Five public datasets have been selected for testing and evaluating the proposed approach. These datasets span from social media insights, to weather information and records of supermarket sales. All five datasets are being presented in subsection III-C. The decision to use multiple datasets, rather than relying on a single set, was made to assess the LLM's ability to understand and operate across various contexts, checking that its capabilities are not limited to a single type of data.
3) Query Design: For each of the five datasets, a set of five queries have been authored, making a total of 25 queries for testing the offline LLM. The queries are categorized as "Basic", "Intermediate", or "Advanced", according to the complexity of the tasks they require the LLM to produce code for. Their definition is the following:

- Basic: These queries will involve straightforward filtering, counting, or retrieving unique values. They shall require simple operations that are fundamental for understanding the structure and content of the dataset.
- Intermediate: These queries will involve grouping, aggregating, or performing basic arithmetic operations. They shall require more complex operations, but will still rely on standard data manipulation techniques.
- Advanced: These queries will involve more complex operations such as multi-level grouping, merging or exploding columns, and statistical analysis. They shall test the LLM's ability to generate code for handling intricate data manipulation tasks, or generate insights from the data.

The 25 authored queries are provided in subsection IV-C.

4) Execution Plan: Each of the 25 queries authored will be provided ten times to the offline LLM. The complete process will be executed, and the results will be recorded. This means that, for each query, ten tests will be conducted. This approach assesses the model's consistency by analyzing code variability, stability, and reproducibility across multiple iterations. Notably, this methodology aligns with practices in experimental research, where repeated tests help better evaluate findings [54]. To sum up, a total of 250 tests will be conducted, ten for each of the 25 queries authored, based on the datasets selected in the context of this study.
5) Evaluation Metrics: Each test will be evaluated based on a series of metrics. With the completion of all 250 tests, the results will be collected, and the metrics will be merged into one main evaluation dataset. Since this study is conducted separately on two offline LLMs, a complete total of 500 tests will be gathered. Each LLM's test is assessed based on the following parameters:
   - Functional Correctness: This is to determine whether the code's produced result is correct or not. In other words, this metric evaluates if the code has properly produced the correct result that the end-user intended to see, based on their natural language query.
   - Readability: This metric provides an overview on the generated code's readability by a human. The score is being calculated by a custom function running in the data processing platform, which can be viewed in the appendix section A. This function evaluates the code by assessing three factors: Line length, method call chains, and nested structures. It penalizes code with lines longer than 80 characters, method call chains longer than 3 calls, and nesting depths greater than 2 levels. The function assigns a score between 1 and 3, with 3 indicating highly readable code.
   - Efficiency: This metric refers to the computational resource efficiency of each query / test to the LLM server. The response time, GPU usage, CPU usage, and memory utilization for both GPU and CPU are monitored using a resource monitoring agent (a Python code snippet). This will help assess the efficiency of the code generation process, in terms of computational costs.
   - Contextual Performance: This is to examine the LLM's performance in different levels of the queries' contextual complexity. As previously outlined, all queries are categorized into Basic, Intermediate, and Advanced levels. The results will reveal how the LLM performs across these three categories, which will also allow for an effectiveness comparison across the levels.
   - Automation: This metric assesses whether the code generated by the LLM could be run automatically, without human intervention, or semi-automatically, with minimal human intervention. In some cases, the LLM might produce code that is accompanied by natural language explanations, despite the prompt's explicit instruction not to include additional text. Any explanatory text should be provided only within Python comments. Moreover, the LLM might define a variable to store the results of the commands, other than the one it was instructed by the prompt to define. In the cases where explanations outside comments were produced, or a clarification as to which variable the final analysis results would be stored, human intervention was needed, in order to isolate/refine the code generated and continue the process.
   - Error Handling: This metric examines if the code provided resulted in errors during its execution. These errors could range from minor issues to more severe problems, that could cause the system to terminate its process.
6) Data Collection and Analysis: The final step in this study's methodology is to gather the results from all 250 tests, merge them into a single dataset, and perform an exploratory analysis. Thus, each LLM will have a dataset of 250 rows. This analysis will provide insights into the LLM's performance across the outlined evaluation areas. The main goal of this study will finally be evaluated: Whether offline LLMs can assist data science, by generating code for data analysis operations.

## B. SYSTEM ARCHITECTURE

Fig. 1 presents an overview of this study's architecture. The process begins with the end-user submitting a natural language query. The relevant dataset is specified in advance, in order to determine the proper dataset summary for loading, as well as the main dataset file for code application in the data processing platform. The user's query is being merged, along with the dataset's summary, under one main message to the LLM, through the prompt engineering step. The prompt, defined in subsection IV-B, combines the query and the summary, also providing essential instructions to the Codestral and Qwen models.

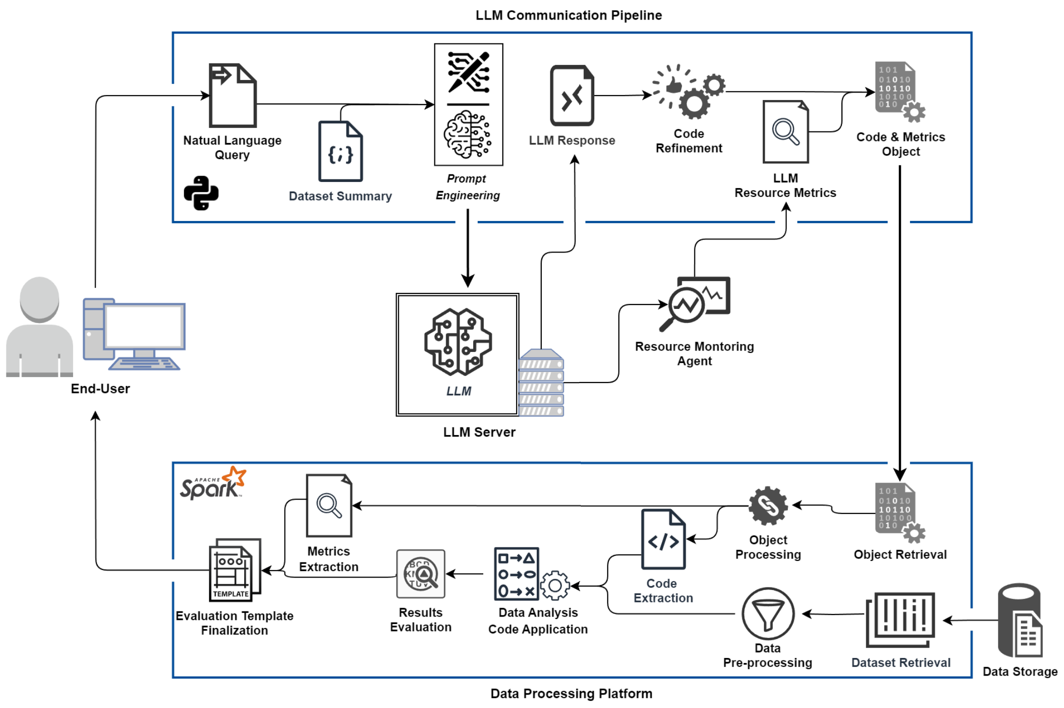

**FIGURE 1.** System architecture overview.

Once an LLM receives this message, the response generation process begins. During this time, a simple python script is monitoring the LLM server's computational resources. Upon completion, the model's response is sent back to the communication pipeline, along with the monitoring results. The end-user proceeds to examine it and evaluates whether the code can be forwarded directly to the next phase (so to confirm automation), or it should undergo minimal modifications (such as removal of additional text that had not been commented by the LLM), thus defining this test as semi-automated and reject automation. After this step, the generated code and monitoring results are being combined into a single JSON object (see Listing 1). The LLM communication pipeline ceases its operation by sending the newly created object to the data processing platform, which triggers the initiation of a new Python Spark job. Once the data processing platform begins operating, the pre-defined dataset is retrieved and pre-processed accordingly. At the same time, the object received by the LLM communication pipeline is parsed, extracting the generated code and the monitoring performance results.

Then comes the generated data analysis code application step, executing the provided commands to the loaded dataset, in order to get the desired analysis results. Upon completion of this step, the analysis results are extracted. A final JSON object is being prepared, as each test's result template, containing the LLM server's monitoring metrics, the generated code, and additional attributes of the process. The object is stored locally, along with the analysis results generated by the code application step. Both files are being

```
submit_spark_job_kpi(
spark_master_ip="<spark-ip-here>",
code_repetition_id=crid,
dataset="netflix",
user_query=user_message,
generated_code= llm_response,
llm_response_cpu=absolute_cpu,
llm_response_memory=absolute_memory,
llm_response_gpu=avg_gpu,
llm_response_response_time=response_time,
llm_response_gpu_mem=avg_gpu_mem,
#llm_version="Codestral V0 1 22B Q6_K",
#llm_version="Qwen 2.5 Coder 14B Q6_K",
output_format="csv",
output_directory="<provided-output-directory"+str(
        repetition_number),
automated="true"
#automated="false
)
```

**Listing 1.** Code from the LLM communication pipeline, preparing to forward the generated code and performance metrics to the data processing platform.

assessed by the end-user, who examines the analysis results and decides whether they are correct and match the desired outcome based on the original query's context. In both cases, a clarification for functional correctness is being added to the test result object (marked as True for a correct result and False for an incorrect one). This step completes the flow of a test, as depicted in Figure's 1 architecture. As previously outlined, for each of the five datasets, five queries will be tested, with ten iterations per query. This whole process will take place twice, once for each offline LLM. Thus, a total of 500 tests will be performed in this study, 250 for each model.

### C. DATASET SELECTION

For this study, five distinct datasets were selected for the evaluation process. A summary for each dataset will be provided to the LLM, as part of the aforementioned prompt message (analyzed in subsection IV-B). The code generated will then be executed using a PySpark job, which will load the dataset and run the LLM's code. Although Spark — the proposed platform's underlying framework — can handle large volumes of data, dataset size was not a criterion for selection. This is because the study focuses on the code generation capabilities of offline LLMs, tailored to data analytics operations, and the approach does not involve loading entire datasets into the model, thus making their size irrelevant. The datasets were chosen from Kaggle and Maven Analytics' Data Playground, both of which offer free and public datasets.

The first dataset selected is the "Netflix: Movies and TV Shows" one, found in Kaggle [55]. It provides extensive information about Netflix's library of movies and TV shows, containing almost 9000 unique entries. Each entry in the dataset is identified by an ID, and the content is categorized by type into either 'Movie' or 'TV Show'. The dataset also includes the title of each entry, along with the director and cast members, thus providing information about each work's the creative teams. Another column indicates where each TV show or movie was produced, categorized by country. More than one country can be listed at one entry. Additionally, the dataset includes the dates that all entries were added to Netflix, along with their year of release, to show when the content was originally made available. A rating column classifies the content according to audience suitability. The duration of each entry is also listed, measured in seasons for TV Shows and minutes for movies. Another column categorizes the content into genres. Lastly, the dataset provides a brief description of each show or movie, offering a snapshot of the plot or premise. This dataset is a resource for analyzing the diversity and trends within Netflix's content library.

The second dataset for selection is the "COVID-19 Twitter" dataset from Kaggle [56]. It includes live tweets related to COVID-19, during three periods: April-June 2020, August-October 2020, and April-June 2021. This dataset includes a collection of almost 1 million tweets, each identified by an ID and their creation dates. A 'source' column indicates the platform (Android or iPhone) used to post the tweet. Both original and cleaned versions of each tweet's text are provided, along with the language used. Metrics, such as the number of favorites and retweets, are also included. In addition, the dataset provides the original author of each tweet, as well as hashtags and user mentions. The location associated with each tweet is recorded in the 'place' column. Last but not least, a set of sentiment analysis scores are included, like compound, negative, neutral, and positive, which lead to an overall sentiment classification column. This dataset can serve as a resource for experimenting with data analysis queries, mainly related to social media platforms.

The third dataset chosen is the "Shared Cars Locations" from Kaggle [57]. It includes over 20 million entries, providing information about the locations of shared cars participating in the AutoTel project, launched in Tel Aviv [58]. The dataset has details such as the latitude and longitude coordinates of each car's location, helping to identify where the cars are stationed throughout the city. It also records the total number of cars available at each location, and provides a list of car identifiers present there. Additionally, the dataset captures the timestamp of each data record (entry), indicating when the information was logged. This dataset can be useful for conducting location-oriented analytics, offering insights into the usage and density of given locations.

The fourth dataset selected is named "Madrid Daily Weather", retrieved from Maven Analytics' Data Playground [59]. It provides almost 7 thousand entries of detailed daily weather information for Madrid, spanning from 1997 to 2015. The dataset includes various weather attributes for each day. It has each entry's date in Central European Time (CET), and key metrics such as maximum, mean, and minimum temperatures recorded in Celsius. It also captures the dew point and its variations, along with humidity levels (maximum, mean, and minimum). In addition, the dataset includes sea level pressure measurements, visibility levels, and wind speed data, including maximum gust speeds. It also lists precipitation levels, cloud cover, and any specific weather events like rain, snow, or fog. The wind direction (recorded in degrees) is also included. This dataset can be useful in analyzing weather-related trends and patterns for a pre-defined geographical area.

Lastly, the fifth dataset chosen is the "Supermarket Sales", once again from Kaggle [60]. Through one thousand entries, it provides information about sales transactions in a supermarket chain, recorded across three different branches over a period of three months. Each transaction is uniquely identified by an invoice ID, and includes details about the branch and city where the sale occurred. The dataset also records the type of customer (Member or Normal) and the gender of the customer. Each product sold is categorized by a product line, and the dataset captures the unit price, quantity sold, and the tax applied. The total amount for each transaction is also provided. Additionally, the dataset includes the date and time of the transaction, the payment method used, and financial metrics such as the cost of goods sold (COGS), gross margin percentage, and gross income. Last but not least, customer ratings of their transactions are also recorded. This dataset can be useful for analyzing sales patterns, customer behavior, and financial performance.

To further ensure the diversity and validity of the proposed evaluation, the five datasets were chosen based on two key criteria: First, they should be open source and publicly available, which promotes reproducibility. Second, they should encompass a variety of domains, which enhances diversity. As previously outlined, all five datasets selected are publicly available. Also, their domains span across entertainment (Netflix: Movies and TV Shows), social media (COVID-19 Twitter), transportation (Shared Cars Locations), meteorology (Madrid Daily Weather), and retail (Supermarket Sales). This heterogeneous selection not only captures the complexity of real-world data analysis tasks, but also aims to eliminate potential biases arising from a limited scope of data [61]. Such a strategy is important for reliably assessing the performance of offline LLMs across different contexts, ensuring that the current approach remains reproducible and unbiased.

The datasets will be used as a basis for the evaluation of the proposed system. Five queries will be selected from each dataset and, expressed in natural language, they will be used to test the selected offline LLM's capabilities on producing code for data profiling and analysis operations. The queries are presented in subsection IV-C.

## IV. TECHNICAL COMPONENTS

As briefly outlined in section III, this article's study consists of an offline LLM, chosen specifically for its code generation capabilities. In addition, a data processing platform has been implemented, in order to efficiently manage the data and apply the code provided by the language model. The communication pipeline between the model and the processing platform consists of a well-designed prompt, part of the prompt engineering efforts conducted during this study. The dataset queries used for testing and evaluation are presented, categorized by dataset and analyzed according to their complexity level.

### A. LLMs AND DATA PROCESSING PLATFORM

The offline LLMs selected during this study are Mistral AI's Codestral [62] and Alibaba's Qwen 2.5 Coder [63]. Codestral is an advanced 22B parameter LLM, designed specifically for code generation. It has been fine-tuned on over 80 programming languages, including widely-used ones like Python, Java, and C++, as well as more specific ones like Fortran and Swift. This diverse training makes Codestral capable of handling a wide variety of coding tasks in different languages and environments. Its performance is benchmarked across various coding tasks, including HumanEval [64], RepoBench [65], and CruxEval [66], where it has proven to excel in both code completion and error minimization, making it a highly suitable choice for both developers and researchers. It currently outperforms other top contenders, CodeLlama 70B [67], DeepSek Coder 33B [68], and Llama 3 70B [69], on almost all benchmark tasks [70]. Also, it supports a large context window of 32k tokens, which is also considerably larger than the current competition. This makes it able to operate on longer and more complex codebases, a very important feature for repository-level code generation and long-range evaluations.

As for Qwen, the Qwen 2.5 Coder is a language model family designed specifically for code generation tasks. The 14B variant of Qwen2.5 Coder stands out for its robust performance across a wide array of coding benchmarks, surpassing even larger models like this study's CodeStral 22B and DeepSeek Coder 33B [68] in several tasks [71]. Built upon the Qwen 2.5 architecture, this model has been fine-tuned on a dataset of over 5.5 trillion tokens, sourced from diverse public code repositories and web-crawled data. It supports multiple programming languages, and demonstrates strong general and mathematical reasoning capabilities. Qwen 2.5 Coder 14B has excelled in benchmarks like HumanEval, MBPP, and BigCodeBench, showing superior results in code generation, completion, and repair. Its extensive training and instruction-tuning have made it effective for both code assistants and real-world applications, offering a balance of power and efficiency. It should also be noted that, in certain benchmarks, the 32B variant of Qwen 2.5 Coder has outperformed even GPT-4o [72]. The selection of the 14B variant over the 32B was based on the hardware specifications of the machine used in this study.

Codestral and Qwen's characteristics lead to a robust performance, setting high standards in code generation with lower latency, a key requirement for developers who are looking for quick feedback during coding. The aforementioned benchmarks demonstrate that both models have performed very well in key areas like Python / SQL code generation, and fill-in-the-middle (FIM) tasks, which are important for completing partial code snippets efficiently. Both have an open weight nature, and can be easily deployed offline under the Mistral AI Non-Production and Apache 2.0 License, hence making them available to research and non-commercial uses without constantly requiring access to the web. This is particularly beneficial for institutions and developers who are concerned about data privacy, among other reasons. The models' combination of speed, accuracy, broad language support, and offline deployment make them a suitable choice for this study's research on an offline code-specific LLM.

Building on the preceding discussion, despite their relatively smaller parameter counts, both models consistently demonstrate superior performance relative to other offline LLMs. Notably, the 7B variant of Qwen2.5-Coder frequently outperforms CodeLlama 70B and, in certain tasks, exceeds the performance of Codestral 22B. Given these observations, comparisons with smaller variants of — already outperformed — models (such as lower-parameter variants of CodeLlama) are deemed to be of limited scientific relevance and may introduce methodological biases. This, combined with the computational resources available for this study, which impose hardware constraints that preclude a comprehensive evaluation of models with more than 30B parameters, enhance the choice of selecting Codestral 22B and Qwen2.5-Coder 14B. Their sizes and performance allow for a rigorous and balanced assessment within the practical limitations of the current experimental setup.

Of course, it should be noted that, given the fast pace of LLM research, future models may surpass Codestral and Qwen, requiring continued evaluation of emerging architectures. This study explores the potential of offline LLMs in data science in general, with Codestral and Qwen being two valuable models for initial research. Since it is not tailored to any specific model, this study can be generalized and applied to future models. As for the reasons behind the selection of an offline LLM — in comparison to online solutions — these reasons vary. First, with offline models, data processing is done locally, increasing data privacy (and security) and ensuring compliance with potential privacy regulations. In addition, this approach helps eliminate the risks associated with sensitive data being sent to external servers. Second, offline LLMs provide customization and control, since models can be fine-tuned to specific tasks and integrated into custom systems, something that is often not possible with commercial cloud-based solutions.

Furthermore, offline LLMs can be cost-efficient in the long run, since no subscriptions are necessary and extended use does not increase their cost, which could be the case with subscription or usage-based pricing LLMs. They also offer reliability and low latency for real-time applications since they are not dependent on internet connectivity. This also ensures that users do not experience interruptions due to potential changes in services by external providers. Also, offline LLMs allow for ethical and regulatory compliance, since they can be tailored to be compliant with local laws and ethical standards. Finally, they promote research and innovation, since they provide a platform for exploring new AI methodologies and applications, without the limitations associated with commercial services.

While online LLMs are often noted for their superior accuracy and faster response times — attributable to their dynamic, cloud-based infrastructures — such advantages were not the primary focus of this study. As already outlined, part of the study's central motivation is to ensure that both data and metadata remain on premises, thereby preserving privacy, upholding regulatory compliance, and maintaining complete control over the processing environment. Although online models might deliver performance gains under specific conditions, their inherent reliance on external servers and network connectivity introduces variables that fall outside the controlled scope of the research.

As mentioned before, the offline LLM will provide its response — generated code — to a data processing platform. There, the code received will be executed and applied to the loaded dataset. For this study, Apache Spark [73] has been selected as the optimal choice for data management and processing operations. Apache Spark is a platform for efficient data processing. It is designed for high speed and efficiency in processing all kinds of data volumes. It supports in-memory computation, which greatly boosts performance, especially for iterative algorithms. This makes Spark easily scalable across clusters and thus ideal for big data processing. Another strength is its versatility, which supports a wide range of different tasks related to data processing, like batch processing, real-time streaming, machine learning, and graph processing.

When combined with Python via PySpark, Apache Spark becomes more accessible. PySpark provides users with the simplicity and flexibility found in Python, so that they can make use of Spark's distributed computing capabilities while coding in a language they are familiar with. This integration makes it easier to perform several operations, like complex data transformations, apply machine learning algorithms, and handle large datasets, all while benefiting from Python's extensive libraries and community support. In this study, code that is received by the LLM is being handled by PySpark jobs, hence executed in the Spark environment. Spark has been the preference of several dig data management proposals and solutions, providing scalable resource management and easy deployment, and thus assisting to optimal data management and analysis across various fields [74], [75].

## B. PROMPT FORMULATION

In this study's offline environment, the primary optimization is achieved through a thoughtfully engineered prompt that communicates detailed instructions to the language model. As the models utilized in this study are Instruct versions, they are consequently designed to depend on well-formulated prompts, aiming towards improved performance and more accurate command generation. Apart from defining the objective and providing essential context to the model, this approach ensures that the output is efficient and tailored to the specific dataset. Hence, the focus on prompt formulation is a deliberate strategy for optimizing the study's offline setting, where traditional optimizations are less applicable.

In addition to optimizing performance, the detailed prompt formulation strategy serves to address the risk of biased outputs, based on the aforementioned provision of explicit instructions and comprehensive context. Combined with the diverse dataset selection — spanning multiple domains — this strategy aims to minimize ambiguity, and limits the risk of unintended consequences. Moreover, the repetitive testing protocol of the study allows for continuous monitoring of the outputs, ensuring that any deviations or potential biases would be promptly identified and addressed.

In light of the above, to enhance the code generation results of Codestral and Qwen LLMs, a carefully crafted prompt message has been developed to initiate communication with the model (Listing 2). Each time an end-user submits a natural language query, it will be processed alongside this prompt message, designed to guide the model's responses for optimal code generation. This approach utilizes the 'few-shot' prompt engineering technique, by providing the model with essential information about the dataset in use, ensuring that the generated code is tailored specifically to that dataset. The prompt includes a summary of the dataset's structure, format, and columns, along with detailed information about each column. This allows the data analysis code generated to be precisely customized for each dataset. The exact prompt message used is provided below:

The outlined prompt message contains a set of instructions towards the language model, but it also provides information (specified as 'context') regarding the dataset to which the generated code will be applied. A breakdown of the key components of the message's structure are as follows:

```
system_message = (
"<s>[INST]Youareavirtualassistantspecializedin
        generatingPythonSparkcommandsbasedonthe
        contextprovidedbelow.Yourtaskistooutputonly
        PythonSparkcommands.Donotaddadditionaltextor
        explanation.Strictlyuseonlycommandsthatcanbe
        appliedtoaSparkDataframe.RefertotheSpark
        DataFrameas'df',andstoretheresultsintheSpark
        Dataframenamed'processeddf'.Context:[/INST]" \
+ str(dataset_summary) \
+ "[INST]Exampleinput:'filterthedatawheretemperatureis
        inthebottom20%ofthetotaltemperaturevalues'.
        Exampleoutput:'processeddf=df.filter(df["
        Temperature"]<=df.approxQuantile("Temperature",
        [0.2],0.0)[0])'.Ensurethatnobackslashesorescape
        charactersareincludedinthegeneratedcommands.
        Generateefficientcommands,usingasfewstepsas
        possible.Separatecommandswithasemicolonor
        newline.Ifyouprovidetextorexplanation,doitusing
        pythoncommenting,with'#'.Iftheinputiscompletely
        incomprehensible,respondwith'LLMERROR:'.This
        shouldberare.Otherwise,generateonlytherequired
        PythonSparkcommands.[/INST]</s>"
)
```

**Listing 2.** A prompt message crafted for this study's communication with the LLM.

- Objective Definition: The prompt engineer should clearly state the goal they want the language model to achieve. In this case, the goal is to generate Python Spark commands without any additional text. This step ensures that the model understands the scope and purpose of its task. It should be noted that the PySpark code generation request adds additional complexity to the LLM, since PySpark DataFrame commands have subtle differences when compared to widely-used Pandas DataFrame commands. The model's ability to differentiate between the two and provide purely PySpark DataFrame commands is also evaluated.
- Context Provision: The prompt engineer should provide the language model with the necessary context to understand the task. This includes summarizing the structure and content of the dataset it will be working with. Through this process, the prompt gives the model necessary background information to generate appropriate outputs, specifically related to a Spark DataFrame referred to as 'df'.
- Instruction Clarity: The prompt engineer should ensure the instructions are clear and precise, by specifying what the model may and may not do. For example, it should only generate Python Spark commands and not any explanatory text. Moreover, it should separate commands with a semicolon or newline. In addition, the prompt explicitly instructs the model to avoid generating additional text, ensuring the output is focused and relevant. It also specifies how to handle errors, and encourages efficiency in command generation.
- Examples Inclusion: The prompt engineer should provide examples to illustrate the desired input and output. This helps the model to understand the format and structure of the expected commands. So, examples are provided to guide the model in understanding the format and nature of the expected output, and thus helping to align its responses with the desired results.
- Error Handling: The prompt engineer should include instructions on what to do if the input is incomprehensible. This ensures the model has a fallback plan, in case it encounters unexpected input. The inclusion of an error handling explanation ensures that the model can respond appropriately if it encounters input it cannot process, enhancing the robustness of the interaction.
- Message Formatting: The prompt engineer should follow any specific formatting requirements for the system they are using, in order to ensure the message is interpreted correctly. The prompt adheres to specific formatting requirements, like the inclusion of '[INST]' commands, to ensure that the language model processes and interprets the instructions correctly, increasing the chances of precise and reliable outputs.

As shown in the middle of Listing 2, the prompt message also incorporates a variable declared as 'dataset_summary'. This variable contains a summary of the dataset that is under evaluation every time. Thus, five dataset summaries have been carefully crafted by a member of the current research team, ensuring that they accurately reflect all aspects of each dataset. Each summary follows a consistent structure that includes an introductory statement about the dataset, a detailed description of each column, including its name, type, description, and sample values, and a note specifying that the sample data is synthetic and intended for illustrative purposes. This well-organized format intends to help LLMs understand the data's structure, easing the process of analyzing it. An example column found in the "Shared Cars Locations" dataset can be seen in Listing 3 below:

```
{
    "name": "timestamp",
    "type": "object",
    "simplified_type": "datetime",
    "description": "TimestampofthedatarecordinUTC.",
    "sample_values": [
        "2019—12—0621:51:02UTC",
        "2019—11—2914:00:02UTC",
        "2019—09—2013:21:03UTC"
    ],
    "datetime_format": "%Y—%m—%d%H:%M:%S%Z"
}
```

**Listing 3.** The 'timestamp' column of "Shared Cars Locations" dataset, as described in the corresponding dataset's summary.

In the outlined example, the 'timestamp' column includes records of the exact date and time when each data entry was logged. It is represented as an object type, with a simplified type of 'datetime' to indicate that it contains timestamp values. The description specifies that the timestamps are in Coordinated Universal Time (UTC), whilst their format follows the pattern "%Y-%m-%d %H:%M:%S %Z" (year-month-day hour:minute:second timezone). The complete dataset summary of the "Shared Cars Locations" dataset can be found in the appendix section B. This detailed level of information is essential for an offline LLM, as it allows the model to load the dataset summary and obtain a complete understanding of the data's structure. Based on these summaries, as part of the complete prompt message, the LLM can better tailor its code generation to handle specific attributes and formats of the datasets.

In short, with this prompt, the model is guided to focus only on generating Python Spark commands, so that it gives output that runs accurately within Spark to perform operations on data. Emphasis is given to clarity and precision, avoiding unnecessary text, and some instructions are included for handling errors or unclear input. This approach aims to improve the interaction with the language model, therefore increasing efficiency and effectiveness in the generation of data analysis code. The outlined prompt message is part of the communication pipeline between the LLM and the Spark-based data processing platform.

### C. DATASET QUERIES

As outlined in subsection III-A, the queries crafted for this study are categorized as Basic, Intermediate and Advanced. Five queries have been created for each of the five datasets, making a total of 25 questions to the offline LLM. Each dataset consists of one Basic, two Intermediate and two Advanced queries. The queries were classified into these three levels based on an assessment of their inherent complexity, in terms of required operations and data manipulation. Since there is not a universally accepted standard for query difficulty categorizations, research has been conducted to establish a three-level classification of query complexity in a safe manner [76], [77].

Although this classification involves a degree of subjectivity, it was iteratively refined to ensure that each category distinctly represents a specific level of challenge for the language model. As previously outlined, to counterbalance any subjective bias, each query is executed multiple times (ten iterations), and performance is evaluated using a series of metrics, including functional correctness, readability, efficiency, contextual performance, automation, and error handling. This structured approach is consistent with experimental methodologies in the field, where tasks are segmented by complexity to evaluate system performance comprehensively [78], [79]. The full list of queries, organized per dataset, can be seen below:

For the "Netflix: Movies and TV Shows" dataset, the following queries have been created:

1) "Count the Number of TV Shows per Country." (Basic): This query tests the LLM's ability to generate code for performing group-by operations and aggregate functions.
2) "Find the Average Duration of only the Movies, in minutes." (Intermediate): This query tests the LLM's ability to perform arithmetic operations and handle different data types (e.g., extracting numeric values from strings).
3) "Return the Top 5 Most Frequent Genres for Movies only, Released After 2010." (Intermediate): This query involves filtering, counting, sorting, and handling lists within a column (genres), adding more complexity to the LLM.
4) "Identify the top five directors who have worked with the greatest number of different actors." (Advanced): This query will test the LLM's ability to generate code for data manipulation, transformation, aggregation, and sorting, in order to identify the top 5 directors.
5) "Provide the top 10 most busy actors in solely American Movies, from 1995 onwards." (Advanced): This query involves filtering data for American movies released from 1995 onwards, transforming the cast column to list individual actors, and then aggregating and sorting the data to identify the top 10 most busy actors.

Regarding the "COVID-19 Twitter" dataset, the following queries have been crafted:

1) "Determine the Number of Tweets Containing User Mentions." (Basic): This query tests the LLM's ability to provide code for filtering rows based on non-null values (the 'user_mentions' column), and count them.
2) "Calculate the top 7 authors with the highest retweet count." (Intermediate): This query involves generating code for grouping data by the original author, summing the retweet counts, and then identifying the top 7 authors with the highest total retweet counts.
3) "Analyze the Daily Tweet Volume Over Time." (Intermediate): This query tests the LLM's capacity to produce code that will perform temporal analysis, by grouping tweets by date and counting the daily volume.
4) "Provide the names of the top 5 users mentioned in tweets, every month (also include the corresponding month and year)." (Advanced): Based on this query, the LLM should generate code that involves converting timestamps to extract the month and year, exploding a column to handle multiple user mentions per tweet, grouping by month and user mentions to count the occurrences, and then identifying the top 5 most mentioned users for each month.
5) "Provide the top 5 weeks (and their years) with the most dense tweets posted, in terms of total clean words included" (Advanced): This query assesses the LLM's ability to generate code for converting timestamps to extract the year and week, calculating the total number of words in each tweet, grouping the data by year and week to sum the word counts, and then identifying the top 5 weeks with the highest total word count in tweets.

For the "Shared Cars Locations" dataset, the queries created are as follows:

1) "Filter Locations with More Than 3 Cars." (Basic): This query tests the LLM's ability to perform basic filtering operations based on numerical data.
2) "Find the Top 5 Locations with the Most Cars Recorded." (Intermediate): This query involves sum-

ming the total number of cars for each location and ranking the results.

3) "List 100 Records at most, from December 2019 to January 2020." (Intermediate): This query tests the LLM's ability to filter data based on date information twice (providing a date window), along with an option to keep only the first 100 entries.
4) "Provide the number of the most dense week, and year, in terms of total cars parked, along with the number of total cars." (Advanced): This query will test the LLM's capabilities in generating code for converting timestamps to datetime, extracting week numbers and years, aggregating the total number of cars parked by week, and identifying the week with the highest total number of cars parked.
5) "Find the total number of (unique) cars that visited each location." (Advanced): With this query, the LLM will be tested in generating code for preprocessing data, expanding lists of cars, grouping based on each car ID, and then aggregating data to determine the total number of unique cars stayed at each location.

When it comes to the "Madrid Daily Weather" dataset, the queries created for evaluation are:

1) "Filter Days of 2006 with Max Temperature Above 30 °C." (Basic): This query will test the LLM's capacity to produce code for performing basic filtering operations based on numerical data (temperature).
2) "Count the Number of Foggy Days per Year" (Intermediate): Based on this query, the LLM should provide code for extracting the year from a date column, filtering the dataset for foggy days based on the presence of "Fog" in the events column, and then grouping the data by year to count the number of foggy days per year.
3) "Calculate the monthly average of mean wind speed and mean sea level pressure, per year" (Intermediate): The LLM code generated by this query involves extracting month information (so testing datetime operations), grouping by month, then by year, and calculating the average values for wind speed and sea level pressure.
4) "Analyze the Yearly Variation in Average Humidity and Identify the Top 5 Years with the Highest Increase in Average Humidity Compared to the Previous Year." (Advanced): With this query, the LLM should produce code that involves extracting the year from a date column, calculating the yearly average humidity, determining the increase in average humidity between consecutive years, and identifying the top 5 years with the highest increase in average humidity, always compared to the previous year.
5) "Analyze the Monthly Variation in Temperature Range and Identify the Top 3 Months with the Highest Average Range." (Advanced): Similarly to the previous query, when provided to the LLM, the model should generate code that calculates the daily temperature range, and then groups the data by month to analyze the monthly variation. This code should identify the top 3 months with the highest average temperature range, testing the LLM's ability to produce code for handling complex calculations, group-by operations, and sorting.

As for the "Supermarket Sales" dataset, the five queries authored are provided below:

1) "Count the Number of Sales per Product Line" (Basic): With this query, the LLM should produce code for performing a group-by operation (for the product line) and one aggregate function (for the total number of sales).
2) "Find the Average Rating for Each Payment Method" (Intermediate): With this query, the LLM should generate code that involves grouping the dataset by payment method, and calculating the average rating for each payment method, slightly increasing the difficulty from the first query.
3) "Find the average quantity of Electronic accessories purchased by card, for each city" (Intermediate): This query tests the LLM's capacity to generate code that involves filtering the dataset for electronic accessories purchased using credit cards, grouping the data by city, and calculating the average quantity of these purchases for each city.
4) "Analyze the Sales Performance by Branch and Payment Method" (Advanced): Based on this query, the LLM should produce code that involves grouping data by both branch and payment method, calculating aggregate sales, and then also calculating the average rating. Thus, the generated code should handle multi-level grouping and aggregation functions.
5) "Calculate the Correlation Between Unit Price and Quantity Sold for Each Product Line" (Advanced): This query will test the LLM's ability to generate code for statistical analysis within grouped data, since the correlation should be calculated between grouped unit price and quantity values, based on all product lines available.

By examining the outlined queries, one can safely conclude that as the complexity of the tasks increases, there is a shift from simple data profiling to more complex data analysis operations. Queries begin by requesting basic code operations from the LLM, but they progressively evolve into more demanding requests. As a result, Codestral and Qwen will be evaluated on their ability to generate code for a series of data analysis operations, all expressed in natural language. Each LLM will be evaluated separately. This will not only test their proficiency in handling diverse and complex tasks, but also their adaptability in meeting the varying levels of analytical challenges.

## V. TESTING AND RESULTS

Testing was conducted on the same physical machine, in order to ensure consistency and eliminate hardware variability. A research team member closely monitored each of the 250 tests, assessing the generated results with meticulous attention to detail. Each query was run precisely 10 times, except some occasions where minor human errors caused the process to terminate unexpectedly. For example, a research member might have forgotten to change the dataset in the test's initial settings, and applied a natural language query intended for a different dataset. In addition, they might have repeated the test without incrementing the test's iteration number, thus overwriting the new results onto the previous test's files. As for the generated code, it was applied to the retrieved dataset using the exec() command, with the Processing Platform's code specifically looking for a ''processeddf'' variable, as this was where the LLM was instructed to store the final analysis results. Intervention to the LLM-generated code occurred only when the model's response included text that was not formatted as Python comments, or when the model failed to store the final analysis results in the instructed ''processeddf'' variable, even though the rest of the code was functional and correct.

## A. PHYSICAL MACHINE SPECIFICATIONS

Below are the complete system requirements of the physical machine used for running the LLM server. Although powerful, the computer has limitations in offline LLM testing, as it cannot fully utilize models like Codestral and Qwen. This means that the LLMs will not be wholly loaded on the GPU. Both the GPU and the CPU will be in-use during the code generation process. Despite the current hardware constraints for more speed and efficiency, the objective of this study is to evaluate Codestral and Qwen's capabilities in code generation, as offline LLMs. Upgrading to more powerful hardware would likely speed up the code generation processes, but this could be a topic for another study.

**Server Hardware Specifications**

**System Information:**

- **Motherboard:** TUF GAMING X570-PLUS
- **Processor:** AMD Ryzen 7 5800X 8-Core Processor
- **Memory:** 32 GiB DDR4
- **Storage:**
  - 1TB NVMe SSD (Samsung SSD 970 EVO Plus)
  - 4TB HDD (ST4000DM004-2U91)
- **GPU:** NVIDIA GeForce RTX 3080 with 10 GiB of dedicated memory

**Software Information:**

- **LLM Server:** LM Studio 0.2.21
- **Large Language Models:**
  - Codestral v01 22B Q6_K
  - Qwen 2.5 Coder Instruct 14B Q6_K
- **Partial GPU Offload:**
  - 17 layers for Codestral
  - 25 layers for Qwen
- **LLM Response Temperature:** 0.7

Codestral v01 22B Q6_K and Qwen 2.5 Coder 14B Q6_K were selected from the Hugging Face hub [80]. Both were deployed using the LM Studio Server [81]. LM Studio supports the deployment of multiple offline LLMs. This, combined with the fact that this study's architecture is not built upon a specific model, confirms that the proposed methodology is model-agnostic and generalizable across different offline LLM platforms, as demonstrated by the evaluations with both Codestral and Qwen (see V-C). The decision to make use of the Q6_K model versions for this study is primarily due to their efficient performances, thanks to 6-bit quantization. These quantized models reduce memory usage and computational demands, without substantially sacrificing accuracy, making them ideal for offline environments where computational resources may be a challenge. The Q6_K versions' ability to maintain high performance, while being efficient on hardware resources, make them a practical choice for the development settings of this study.

As for the LLMs' response temperature setting to 0.7, this choice was made because the model would balance between creativity and reliability, which is important when evaluating a model's code generation capabilities. Overall, moderate temperature settings help in generating outputs that are both varied and contextually appropriate [82]. A temperature of 0.7 allows the model to generate diverse outputs, avoiding overly deterministic responses that could limit exploration of alternative coding solutions, while still maintain enough coherence to ensure the generated code is functional and relevant.

## B. TESTING RESULTS COLLECTION

Each test generated a set of information, which was organized into a single testing result object, as illustrated in Listing 4. For each of the two LLMs, a total of 250 such objects were combined into a dataset of 250 rows, with each row corresponding to an individual test object. Each row in the final dataset contains detailed information about the associated test, including attributes related to the correctness, readability, and execution performance of the generated code, as well as resource monitoring metrics of the system. The subsequent bullet list explains each attribute in detail, offering a comprehensive overview of the data collected during the testing process.

- ''correctness'': This column includes values 'True' or 'False', assessing if the code has properly produced the result that the end-user intended to see, based on the natural language query provided.
- ''readability'': This column contains numerical values that represent the generated code's readability level by a human. The values it contains are calculated by a custom — yet simple — readability calculation function, producing scores between 1 and 3, with 3 indicating highly readable code.
- ''code_execution_errors'': This column contains information about potential errors that occurred during the generated code's execution. If no errors occurred, the value will be 'None'. If an error did occur, this

column will contain the error message, explaining the malfunction.

- "executed_command": This column contains the full code executed for each record/test. It might also include Python comments.
- "code_repetition_id": This column contains each record's/test's iteration number. Since each query is tested ten times, this column will contain values between 1 and 10, indicating which repetition each entry represents.
- "dataset": This column consists of the names of the five datasets selected in this study, with each entry containing the name of a single dataset. The dataset names are 'supermarket', 'netflix', 'shared-cars-locations', 'covid19-twitter', and 'madrid-daily-weather'. The dataset name in each entry indicates that the corresponding information refers to a test conducted on that specific dataset.
- "user_query": This column contains the exact query submitted by the user and sent to the offline LLM for code generation. The corresponding code generated by the LLM for each query can be found in the "executed_command" column.
- "llm_response_cpu": This column consists of the CPU usage percentages measured during the LLM's code generation process.
- "llm_response_memory": This column provides the memory utilization values of the LLM server during code generation. Once again, the values are percentages.
- "llm_response_gpu": This column contains the GPU usage percentages measured on the LLM server, during the code generation process.
- "llm_response_gpu_mem": This column consists of the percentage values of the GPU memory's utilization, during the LLM's code generation process.
- "llm_response_response_time": This column lists all the response times of the LLM server, from the moment it received the query to the moment it returned the generated response. The values are measured in seconds.
- "automated": This column contains 'True' or 'False' values, to determine whether each entry's/tests code was executed automatically, without human intervention (so 'True'), or semi-automatically, with minimal human intervention (so 'False').
- "query_no": This column contains values that indicate which query number each entry represents. Since five queries were created for each dataset, the possible values for each entry of this column are 'q1', 'q2', 'q3', 'q4', and 'q5'.
- "query_level": This column indicates the contextual complexity level of each entry's query. Since the queries created are categorized as 'basic', 'intermediate', or 'advanced', these are also the possible values of this column.

```
{
"correctness": true,
"readability": 3,
"code_execution_errors": "None",
"executed_command": "#CountthenumberofTVShowsper
        country\n\nprocesseddf=df.filter(df['type']==\"TV
        Show\").groupBy('country').count()",
"code_repetition_id": "7",
"dataset": "netflix",
"user_query": "CounttheNumberofTVShowsperCountry",
"llm_response_cpu": 34.45,
"llm_response_memory": 0.01,
"llm_response_gpu": 20.25,
"llm_response_gpu_mem": 89.03,
"llm_response_response_time": 19.41,
"automated": "true"
}
```

**Listing 4. A testing result (JSON object) from the 'Netflix' dataset, and the basic query 'Count the Number of TV Shows per Country'.**

## C. EVALUATION

The evaluation was conducted using the collected results and the dataset derived from them, following the main evaluation criteria outlined in subsection III-A. The analysis led to several observations and suggestions. A summary of the findings and discussion on future steps will be presented in section VI. It is important to mention that the Contextual Complexity metric has been analyzed in combination with all other evaluation metrics. This approach allows for a better examination of query complexity along with other factors, leading to more informative conclusions. As a result, a separate subsection for Contextual Complexity has not been included.

### 1) FUNCTIONAL CORRECTNESS

The goal of this metric is to assess the success rate of the code generated by the tested LLMs. It evaluates whether the code produces the intended results, based on user queries. The first analysis is to calculate the proportion of correct vs. incorrect outputs, thus examining the overall success rate. The second task is to analyze the correctness by dataset and query level. Then, the correctness rates across all levels of query complexity are presented. The later two tasks comprise a more detailed correctness analysis, expanding the success rates on other factors of the study. All three tasks are presented below.

Based on Fig. 2, the number of correct results from the code generation and application process significantly surpasses the number of incorrect ones. More specifically, for Codestral, 228 out of 250 tests produced a correct output — a good sign of its code generation capabilities — while only 22 tests failed to yield the correct results. In parallel, the evaluation of Qwen 2.5 Coder revealed that 223 tests generated correct outputs, with 27 tests not meeting the expected outcome. This comparison suggests that both LLMs are able to understand the context of user queries, and produce viable code, with Codestral exhibiting a marginally higher success rate. For this process, human observation and validation deemed

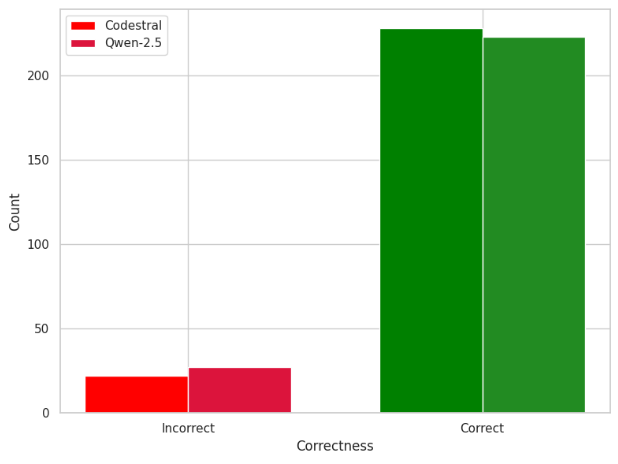

FIGURE 2. Functional correctness of the LLM's generated code plot.

necessary, in order to ensure accuracy. Each time a test was completed, a member of the research team validated the data analysis' output, checking whether it indeed aligned within the scope of the original user query. For each query, Python code has been developed, using the Pandas library to replicate the results. The outputs of these validation scripts were then compared against the outputs generated by the respective LLM.

Fig. 3 compares correctness scores across all five datasets. The performance across the datasets is consistent, suggesting that the LLMs are capable of producing correct code regardless of the context of the data to which its code is applied. For Codestral, the 'Shared Cars Locations' dataset has the lowest correctness rate at 0.86 (43 out of 50 tests producing correct outputs), with the 'COVID19-Twitter' dataset following at 0.88 (44 correct outputs out of 50), while the 'Madrid Daily Weather', 'Netflix', and 'Supermarket Sales' datasets each attain a correctness rate of 0.94 (47 out of 50 tests). In parallel, Qwen-2.5 Coder shows a somewhat different performance profile: Netflix dataset registers the lowest correctness at 0.82 (with 41 out of 50 tests being correct), closely followed by the 'COVID19-Twitter' dataset at 0.84 (42 out of 50 correct tests). The 'Madrid Daily Weather' dataset records a 0.92 rate (46 out of 50 tests producing correct results), whereas both the 'Shared Cars Locations' and 'Supermarket Sales' datasets achieve the top score of 0.94 (with 47 out of 50 correct tests each). Thus, both LLMs demonstrate the capability to generate correct code across varied contexts, with subtle performance differences across datasets.

Fig. 4 illustrates how correctness varies with query complexity. For Codestral, advanced queries yield the lowest correctness rate at 0.84 (84 out of 100 tests produced correct outputs), intermediate queries achieve a rate of 0.95 (95 correct outputs out of 100), and basic queries exhibit the highest accuracy at 0.98 (with only 1 incorrect output in 50 tests). In a similar vein, Qwen 2.5 Coder shows

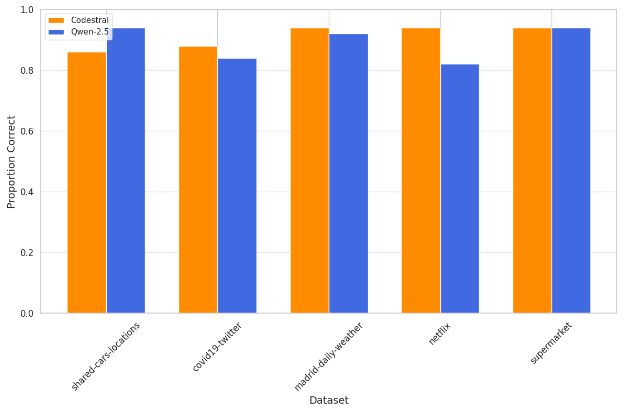

FIGURE 3. Plot depicting the functional correctness, by dataset.

a comparable trend: Advanced queries slightly drop to a correctness rate of 0.83 (83 out of 100 tests being correct), intermediate queries further decrease to 0.91 (91 out of 100 correct outputs), while basic queries maintain the high performance at 0.98 (49 out of 50 correct tests). It is important to note that the number of basic query tests is half that of the advanced and intermediate tests. However, the consistent pattern observed across both models, where increased query complexity leads to a higher frequency of incorrect outputs. This suggests that each LLM tends to struggle more with advanced queries, which is expected to some extent.

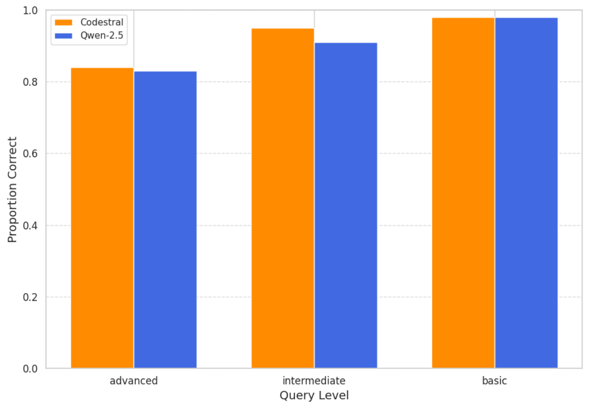

FIGURE 4. Plot for the functional correctness scores by the queries' complexity levels.

### 2) READABILITY

The purpose of this evaluation metric is to assess the readability of the code generated by each LLM, which may influence how easily a human can comprehend the code. The analysis includes three tasks, beginning with a distribution of readability scores across all tests. In addition, the average readability scores filtered by dataset and query complexity level are presented. The analysis approach is similar to the Functional Correctness' one analyzed before. As outlined in subsection III-A, the readability score was calculated by a custom readability function, producing scores ranging from 1 to 3, with 3 suggesting easily readable code. The readability function evaluates code based on line length, method call chains, and nested structures, penalizing code with lines longer than 80 characters, method call chains longer than 3 calls, and nesting depths greater than 2 levels.

Based on the results illustrated in Fig. 5, the majority of tests with Codestral yielded a readability score of '2'. Specifically, 176 out of 250 tests were assigned a readability score of '2', while the remaining 74 tests achieved the top score of '3'. No tests received a score of '1'. In a comparable evaluation, Qwen 2.5 Coder exhibited a similar pattern with 177 tests having a readability score of '2', and 71 tests scoring '3'. However, 2 tests of Qwen were assigned with a score of '1'. Although a very small — and potentially trivial — amount, it could indicate that, on rare occasions, the code generated by Qwen could have more pronounced readability issues. For both models, the primary reason for not attaining the highest score was the presence of long lines of code. This indicates that their code exceed the recommended 80-character limit, a factor that the readability function penalizes because such lines are deemed harder to read and maintain. While the LLMs often generated code with longer lines, which can sometimes enhance code efficiency by combining commands, it tends to compromise human readability. This is the main reason why most of the generated code did not achieve the highest readability score of '3'.

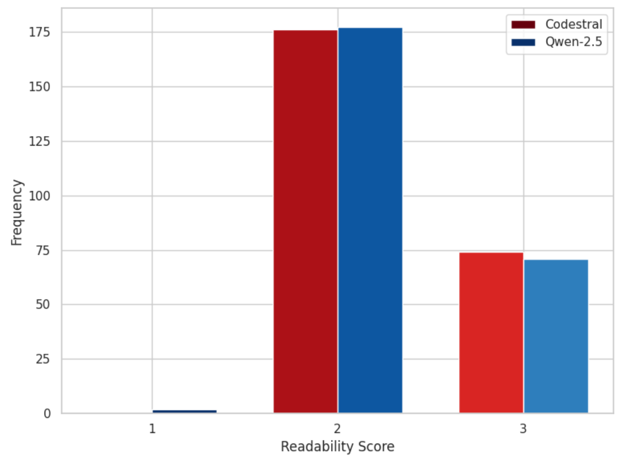

**FIGURE 5. Plot for the distribution of readability scores across the tests conducted.**

Fig. 6 illustrates the average readability scores by dataset. For Codestral, the 'Madrid Daily Weather' dataset registers the lowest average readability score at 2.08, where 46 out of 50 tests received a score of '2'. On the same dataset, Qwen 2.5 Coder yields a slightly higher average of 2.16, with 42 out of 50 tests having a score of '2', suggesting marginally better optimal formatting. Similarly, the 'Shared Cars Locations' dataset shows Codestral's average at 2.28 (with 36 out of 50 tests rated as '2'), while Qwen records an average of 2.38 (with 31 out of 50 tests having a '2' rated code). In the case of the 'Netflix' dataset, Codestral achieved an average score of 2.30 (with 35 out of 50 tests scoring '2'), in contrast to Qwen's lower average of 2.16, with 38 out of 50 tests' code rated as '2', but also 2 tests with code rated as '1'. The 'Supermarket Sales' dataset attains similar averages between the two models, 2.38 for Codestral and 2.40 for Qwen. As for the 'COVID19-Twitter' dataset, Codestral leads with an average of 2.44, compared to Qwen's 2.28. Although the results do not allow for safe conclusions or suggestions, they could suggest that operations in some datasets can be more efficient (in terms of the number of commands used), potentially due to the nature and context of the data within these datasets.

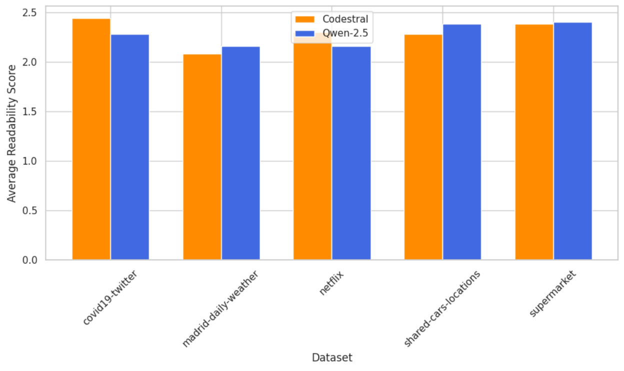

**FIGURE 6. Plot depicting the average readability scores by dataset.**

Fig. 7 presents the average readability scores by query complexity levels. For Codestral, basic queries achieve the highest average readability of 2.8, since 40 out of 50 tests included code with a readability score of '3'. Intermediate queries average 2.26, with 74 out of 100 tests scoring '2', and advanced queries have the lowest average at 2.08, as 92 out of 100 tests were credited with readability of '2', and thus only 8 tests with readability of '3'. In contrast, Qwen 2.5 Coder's basic queries average a readability score of 2.58, with 29 out of 50 tests having code with a readability score of '3'. Intermediate queries score 2.29 average readability, with 69 out of 100 tests having code of '2', but also one test with score of '1'. Advanced queries score 2.11, which corresponds to 87 out of 100 tests with code readability of '2', 12 with score of '3', and one test with score of '1'. These results collectively suggest that both LLMs tend to produce less readable code as query complexity increases, which is expected. It should be noted that, while Codestral appears to generate particularly concise, single-line code for basic queries, Qwen's approach for simpler queries seems to be marginally less efficient. For intermediate and advanced queries, both models produced code that generally consisted of longer lines of commands, which contributed to the higher frequency of readability scores of '2'.

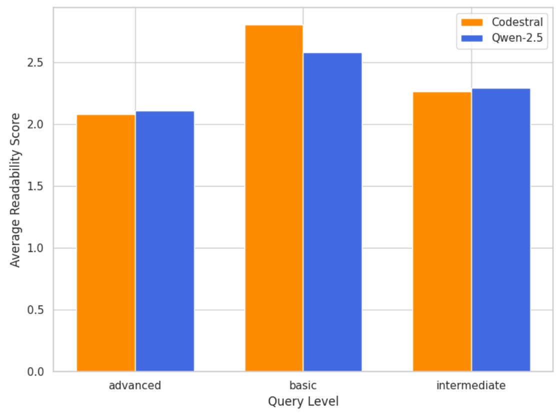

**FIGURE 7.** **Plot displaying the average readability by query complexity level.**

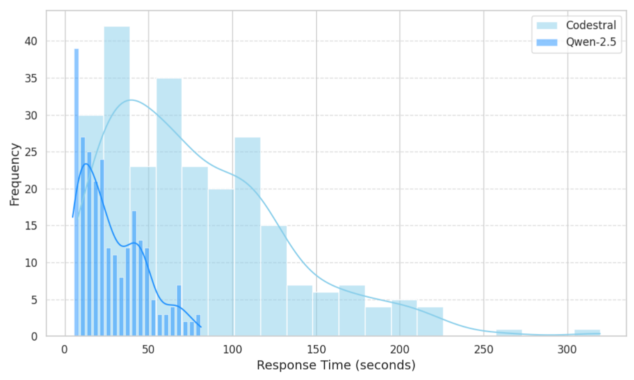

**FIGURE 8.** **Distribution of the LLM server's response time during code generation.**

### 3) EFFICIENCY

This metric aims to evaluate the LLMs' performance in terms of computational resource usage, during the code generation process. As outlined in subsection V-A, the Codestral and Qwen 2.5 Coder models are not fully loaded on the GPU. This means that both the GPU and the CPU will be in-use during the code generation process, especially the GPU's memory. The first analysis task is to assess the GPU usage, CPU usage, GPU memory, system memory (RAM), and response time distributions across the tests conducted for each LLM. The second task is to inspect the computational resources by query complexity and readability, in order to see how is the performance affected across all levels of query complexity, as well as the two levels of readability, keeping in mind that readability scores of '2' mean longer lines of code (and thus longer periods of code generation).

Fig. 8 depicts the distribution of response time (in seconds) during the LLM server's code generation process. Response time is measured from the moment the natural language query is sent to the LLM, until the complete response is returned. As illustrated in the figure, most code generation processes had response times under 100 seconds, for both LLMs. In the case of Codestral, the mean response time was 79.64 seconds, with a median of 67.90 seconds, and some instances exceeding 200 seconds (reaching up to a maximum of 319.53 seconds). Qwen 2.5 Coder demonstrated improved efficiency, recording a mean of 27.66 seconds, a median of 22.98 seconds, and a maximum response time of 81.38 seconds. This noteworthy difference suggests that, whereas Codestral's slower performance is mostly attributable to the partial offloading of operations to the GPU, Qwen benefits from a larger layer GPU offload to, since it has fewer parameters (14B compared to Codestral's 22B), and thus a smaller total size. This allows Qwen to achieve response times that are closer to real-time standards.

Figures 9 and 10 showcase the distribution of CPU and memory usage (in percentages) for the LLM server, during the code generation process. For Codestral, the majority of queries resulted in an average CPU usage of approximately 28.5%. Only a few processes exceeded 50% usage levels, indicating a relatively moderate CPU demand during code generation. In contrast, Qwen 2.5 Coder exhibited an average CPU usage of around 37.6%, with its interquartile range suggesting that many processes consumed roughly between 27% and 47.8% of the available CPU resources. Regarding system memory, both models demonstrated minimal RAM utilization. Codestral's memory usage averaged around 0.44%, with nearly all queries falling within the 0–2% range, while Qwen's average was similarly low at approximately 0.45%, though its distribution was slightly tighter (median of 0.48% and a maximum of 4.65%). These observations imply that, despite the moderate CPU demand, particularly in Qwen's case, the system also utilized other resources, which is expected, since both LLMs were deployed across both GPU and CPU.

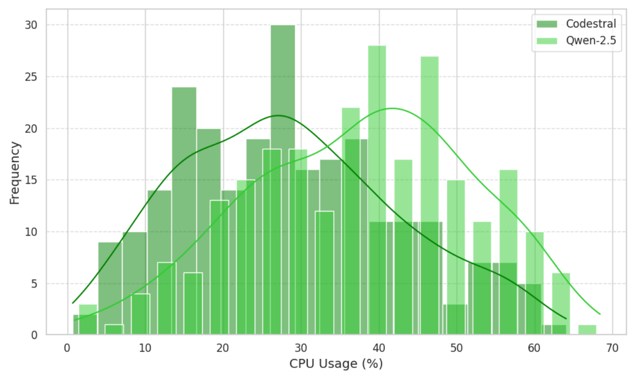

**FIGURE 9.** **Distribution of the LLM server's CPU usage during code generation.**

Figures 11 and 12 provide further insights into resource utilization, as they illustrate the overall GPU distributions (in percentages) for all tests conducted during the LLM server's code generation processes. For GPU usage, most processes required roughly 20% of the GPU's capacity. In the case of Codestral, the average GPU usage was approximately 21.3%, with the interquartile range spanning from about

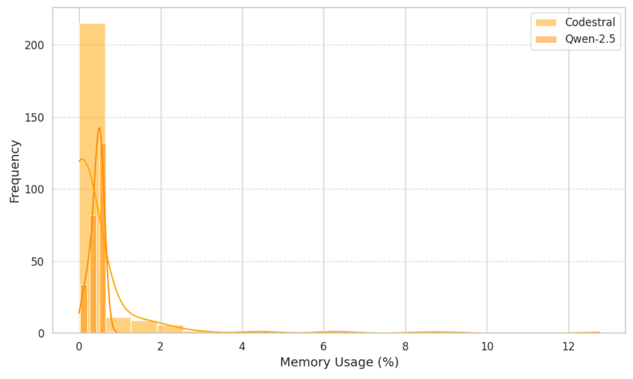

**FIGURE 10. Distribution of the LLM server's memory usage during code generation.**

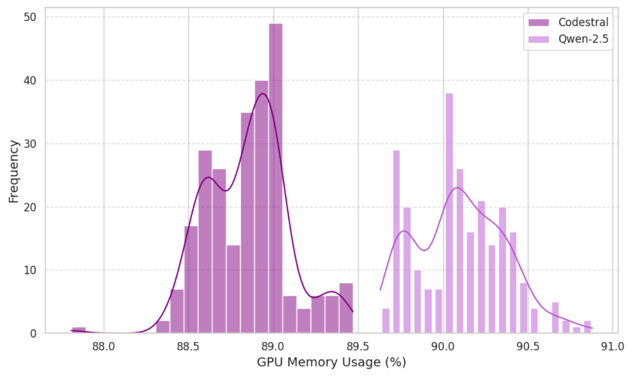

**FIGURE 12. Distribution of the LLM server's GPU memory usage during code generation.**

18.7% to 21.8%. As for Qwen 2.5 Coder, it exhibited a slightly lower mean GPU usage of 18.0%, with most tests falling between 14.4% and 19.3%. These moderate usage levels align with the CPU utilization observed, given that both models operate across CPU and GPU resources. In contrast, GPU memory usage was consistently high for both models. Codestral utilized on average about 88.8% of the available GPU memory, whereas Qwen leveraged an average of approximately 90.1%. This indicates that a substantial portion of the models' parameters were loaded in GPU memory. The combination of moderate CPU and GPU utilization, alongside high GPU memory utilization, could suggest that the code generation processes likely required short bursts of processing power, rather than continuous, high-intensity computation. The system seemed to maximize GPU memory usage, while distributing computational tasks between GPU and CPU. In addition, the moderate GPU usage levels could indicate that the code generated was not highly complex for the LLMs' capabilities, leading to lower computation needs.

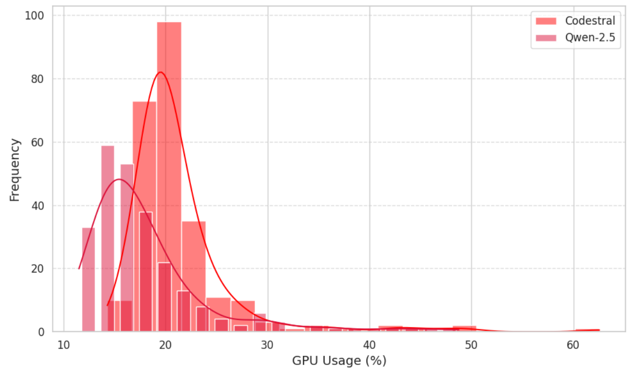

**FIGURE 11. Distribution of the LLM server's GPU usage during code generation.**

Figures 13 and 14 present the average response times by the generated code's readability and by query complexity level, respectively. Regarding readability, for Codestral, tests with a readability score of '2' required significantly more time, averaging about 93.7 seconds per query, whereas those with a score of '3' were generated in roughly 46.2 seconds. In parallel, Qwen 2.5 Coder exhibited a similar trend, but with overall lower response times. Tests with a readability score of '2' averaged about 31.6 seconds, while those rated as '3' were completed in approximately 18.1 seconds. This behavior is expected, as a readability score of '2' indicates the presence of longer lines of commands that demand additional processing time, whereas a score of '3' suggests more concise code leading to faster generation. As for Qwen's faster response times, they can be attributed to the outlined characteristics of its smaller size (14B parameters) and therefore greater layer offloading to the GPU, compared to Codestral.

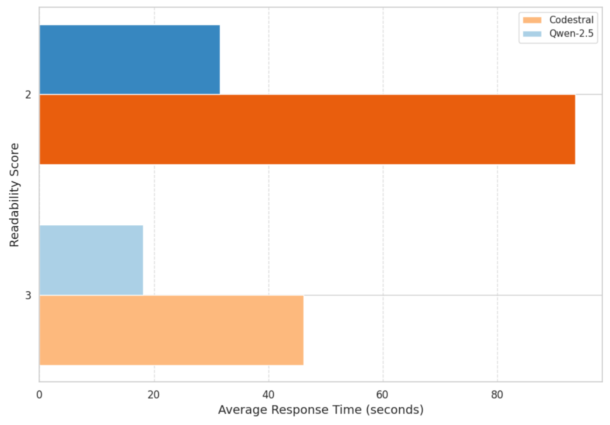

**FIGURE 13. The LLM server's average response time, by code readability.**

In terms of response times based on query complexity, the results also align with expectations. For Codestral, basic queries had the lowest average response time at around 26.5 seconds per query, intermediate queries took roughly 67.5 seconds, and advanced queries required nearly 118.3 seconds per query. Similarly, Qwen 2.5 Coder demonstrated a consistent trend, with basic queries averaging about 10.3 seconds, intermediate queries around 23.6 seconds, and advanced queries approximately 40.5 seconds. These

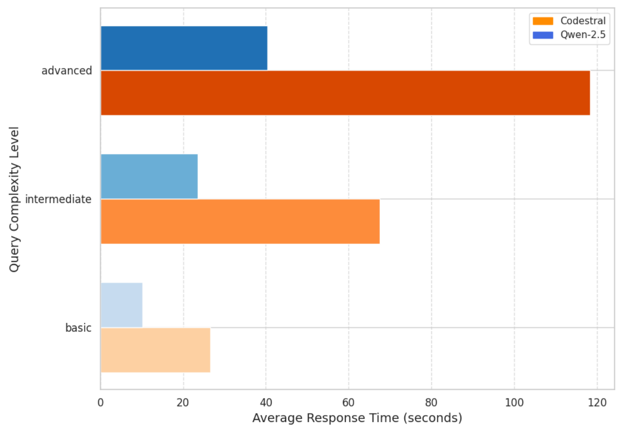

FIGURE 14. The LLM server's average response time, by query complexity level.

findings are reasonable, since basic queries typically result in simpler code that is generated quickly, while the increasing complexity of intermediate and advanced queries naturally leads to longer generation times.

Figures 15 and 16 depict the average CPU and GPU usage levels by readability and query complexity. An interesting observation emerges from both plots. The CPU and GPU usage levels for the two readability scores are mostly comparable. When examining CPU usage, Codestral's values for readability score '2' range from about 27.5% to 32.3%, while for readability score '3' they range from approximately 23.3% to 32.6%. A similar trend is observed for Qwen 2.5 Coder. Under readability score '2', Qwen records around 34.0% to 37.6%. Under readability score '3', the figures range from roughly 41.6% to 42.6%. The two Qwen tests that produced readability score of '1' are not taken under consideration, since the amount could be considered as insignificant. Regarding GPU usage, Codestral's readings for readability score '2' vary from about 20.2% to 21.5%, and for readability score '3', from 20.3% to 22.4%. In the case of Qwen, GPU usage under readability score '2' averages 15.9% to 18.4%, while for readability score '3' the values are in the range of 17.4% to 20.98%. Overall, the usage levels between the two readability groups remain similar.

The results can be attributed to the system's behavior. As the code generation process unfolds, the system appears to reach a steady state in terms of CPU and GPU utilization. Once this steady state is achieved, resource consumption remains constant until the process is complete, regardless of variations in the generated code's length or complexity. This explanation could partially be applied to another observation, regarding query complexity levels. The data indicate that, in some cases, basic queries have higher average CPU and GPU usage than intermediate or advanced queries, sometimes even the highest of the three.

Because the code generation process for basic queries is — usually — significantly shorter (see Fig. 17), their CPU / GPU usage is more influenced by the system's initial performance

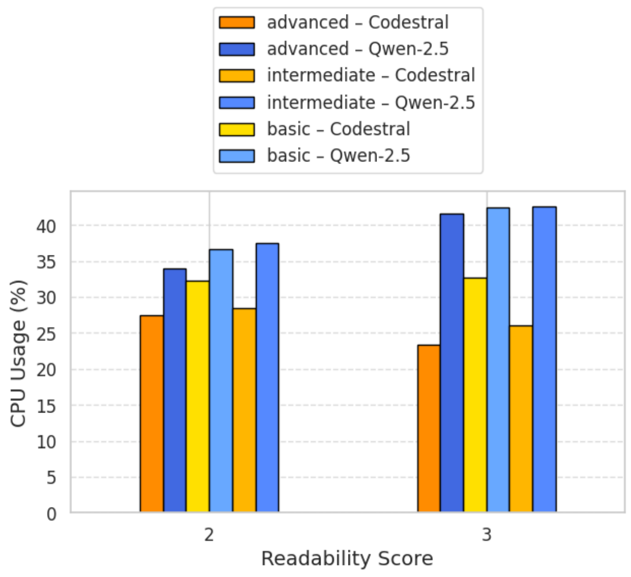

FIGURE 15. Average CPU usage of the LLM server, by readability and query complexity.

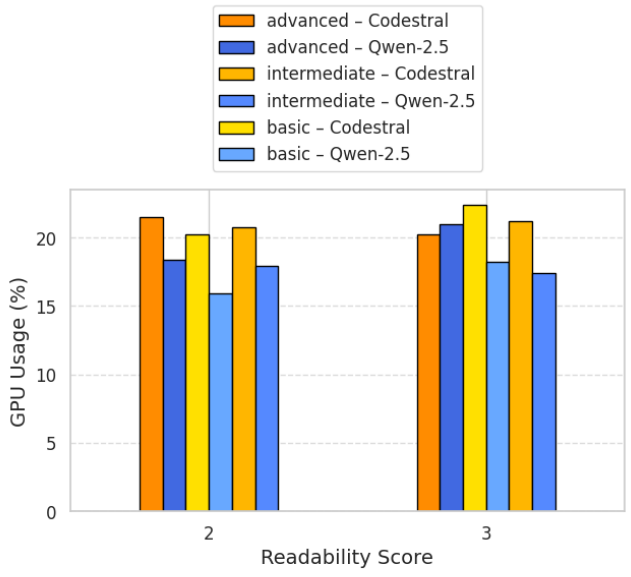

FIGURE 16. The LLM server's average GPU usage, by readability and query complexity.

boost. For intermediate queries, this initial boost may have a smaller impact, since the longer duration allows the system to 'normalize' those early high values. The same could be applied to advanced queries. As time progresses, the system reaches a phase where CPU and GPU usage stabilizes, with only the GPU memory continuing to experience heavy usage. This claim could be further justified by the average GPU memory usage depicted in Fig. 18, where all readability scores and query complexity levels have the same, constant, high demand for GPU memory. However, it's important

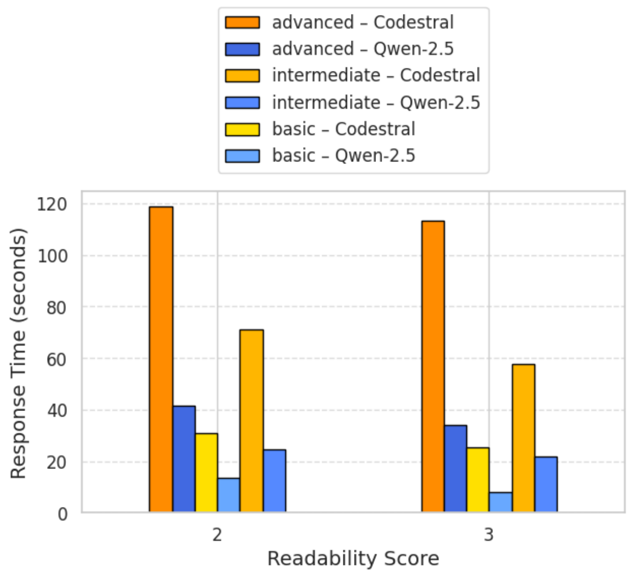

**FIGURE 17.** Average response times of the LLM server, by readability and query complexity.

to note that the differences in values across all cases are relatively small, so these assumptions are made with caution.

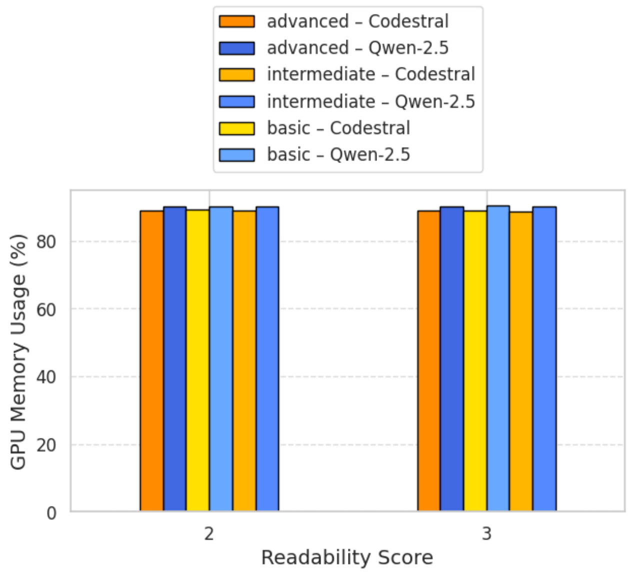

**FIGURE 18.** Average GPU memory usage of the LLM server, by readability and query complexity.

### 4) AUTOMATION

The purpose of this evaluation metric is to determine the extent to which the generated code can be executed automatically, without human intervention. After each code generation process by the LLMs, the response is assessed by the end-user. If the generated code is ready for direct application to the data, the automation assessment is marked as True. However, if minimal human intervention is needed, in order to isolate or slightly refine the code (but the code's functionality produces the desired result), the automation assessment is marked as False. The first analysis task is to observe the amount of tests with positive vs. negative automation labels in their generated code. Next, automation status will be compared with the functional correctness results and the query complexity levels. These analysis tasks will help draw conclusions about the LLMs' ability to provide code directly as it was instructed to through the prompt message, in order to establish an automated pipeline for the data analysis workflow.

Fig. 19 depicts the categorization of all 250 tests based on their generated code's automation attribute. Specifically, for Codestral, 202 out of 250 tests had code which could be forwarded directly to the data processing platform without the need for human intervention, thus marked as 'True' for automation, while the remaining 48 tests included code that required minimal human intervention before execution, thus marked as 'False' for automation. This indicates that the Codestral offline LLM achieved a positive automation rate of about 80%, meaning that only 1 out of 5 tests (or 20%) required human intervention in order to be fully functional. As for Qwen 2.5 Coder, it demonstrated even more promising results, with 239 tests marked as 'True' (approximately 95.6%) and only 11 tests (about 4.4%) requiring minimal intervention. This could indicate that the current version of Qwen is more capable of being used at a fully automated environment, whereas Codestral could benefit from potential future optimizations in its pre-trained configuration.

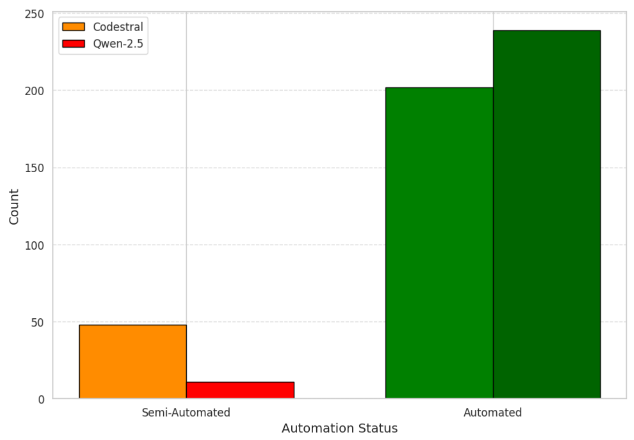

**FIGURE 19.** The amount of automated and semi-automated tests, based on human intervention to their code.

It should be reminded that, as already stated, there are two reasons for human intervention in the generated code. First, the LLM had a tendency to produce additional text that was not formatted as Python comments, despite prompt instructions to eliminate such text. In these cases, a research team member would clean the code from such text, keeping only Python commented information (if present). Shall the process had continued with the explanatory text included, the flow would terminate abruptly during the code application step, as the 'exec()' function would not comprehend how to handle it. Second, the code application step expects the final analysis results to be outputted to a variable named "processeddf". On occasions where the LLM failed to do this, a research team member would redirect the final output to a variable named "processeddf", and continue the flow. Shall the system had not received such a variable in the exec() command, it would be unable to proceed to the results extraction step. Although instructed to output the final result of generated commands to a "processeddf" variable, the LLM sometimes failed to do so.

Figures 20 and 21 illustrate the relationship between automation results and functional correctness, as well as query complexity levels. Regarding the former, it is evident that the correctness of the produced results does not impact the automation of the code they were generated from. Whether or not human intervention occurred, the percentages of correct results remained consistent. Specifically, for Codestral, 75% of tests (meaning 187 out of 250) produced the desired result when the code was automated, while an additional 16.5% (41 out of 250) produced the proper result when the code required minor refinements. In a similar vein, Qwen 2.5 Coder achieved 213 correct outcomes from its fully automated tests, which translates to an 85% success rate, with additional 10 tests (only 4%) that yielded the proper result after minor interventions.

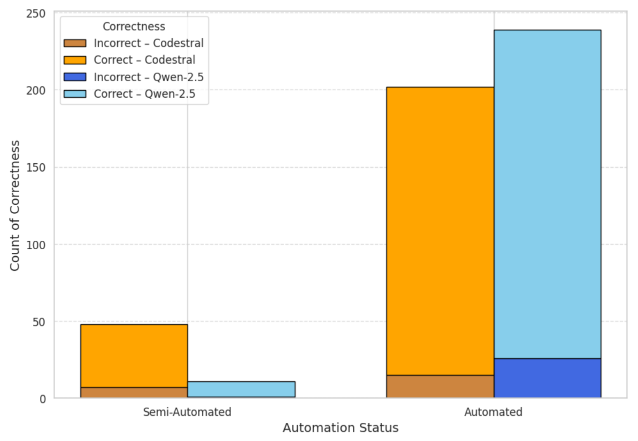

**FIGURE 20. Comparing Functional Correctness results with Automation occurences.**

As for the latter figure, the automation percentage in basic queries for Codestral was 86% (43 out of 50), accompanied by intermediate queries with the same percentage (86 out of 100), while advanced queries had an average automation rate of 73% (73 out of 100 tests being fully automated). In contrast, Qwen 2.5 Coder demonstrated higher automation across all query complexities, with 100% of basic queries (50 out of 50), 91% for intermediate queries (91 out of 100), and 98% for advanced queries (98 out of 100) being fully automated. Although the differences in percentages for Codestral are relatively small, they suggest that Codestral

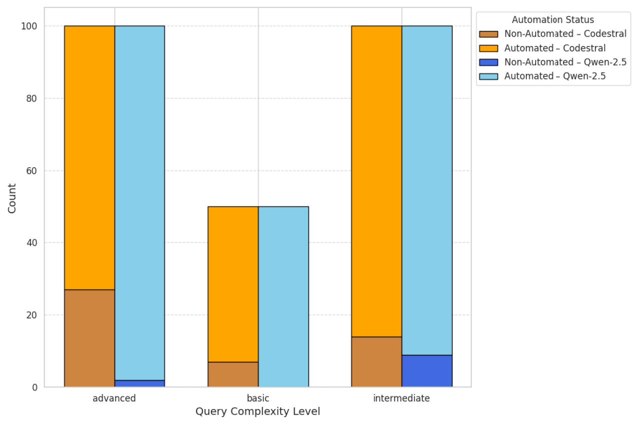

**FIGURE 21. Presenting the automation occurrences by the query complexity levels.**

may tend to produce more additional text as the complexity of queries increases, adding explanations to the commands generated. However, the percentage differences are not significant enough to draw definitive conclusions. Overall, regardless of query complexity, automation levels remained high for both models, emphasizing their effectiveness in generating code that is ready for execution with minimal human intervention.

### 5) ERROR HANDLING

The goal of this evaluation metric is to evaluate the generated code's robustness, determining whether the error levels were high and if they significantly impacted the produced results. The three plots presented below illustrate the distribution of errors across all tests conducted during this study. These plots are categorized by dataset and query complexity level, as well as the potential impact errors had on the functional correctness of the results. Overall, based on Fig. 22, only 20 out of 250 tests (for Codestral) included code with errors, which is a highly positive indicator of the LLM's capabilities, resulting in a 92% success rate. Taking functional correctness into account, 218 out of 250 tests included error-free code that produced the correct result, leading to an 87% success rate.

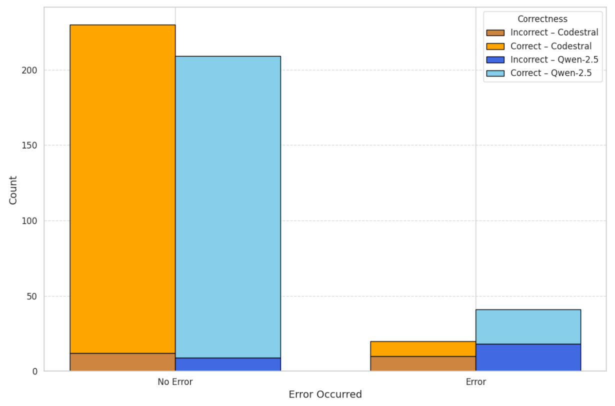

**FIGURE 22. Comparing Functional Correctness results with error counts.**

In comparison, Qwen 2.5 Coder exhibited a somewhat higher error incidence, with 41 out of 250 tests including code with errors, which translates to an 85% success rate. When considering functional correctness, 200 out of 250 tests produced error-free code that yielded the correct result, corresponding to an 80% success rate. This outcome is particularly encouraging, for both LLMs, especially given the fact that the goal was to generate code in PySpark using Spark DataFrames, whose intricate differences from the widely-used pandas DataFrames could have posed challenges for the model, making the low error rate an even more significant achievement. However, it should be noted that, in Qwen's case, an additional statement to the original prompt message had to be included, for the model to properly generate PySpark commands, which it sometimes merged with invalid pure Python ones. This will be further mentioned in subsection VI-A.

Figures 23 and 24 present the distribution of the error occurrences and their impact on functional correctness across all five datasets used for testing, as well as across the three levels of query complexity. Although the amount of errors makes it unwise to draw safe assumptions, some early insights can be formed, providing a basis for future research and analysis. Regarding the distribution per dataset, for Codestral it is the 'Shared Cars Locations' dataset that exhibits the highest number of error-labeled tests, with a total of 13 errors. This significantly surpasses the other datasets, with the 'COVID-19 Twitter' dataset containing 3 errors, the 'Supermarket Sales' dataset 2 errors, and both the 'Madrid Daily Weather' and 'Netflix' datasets having only 1 error each. In comparison, Qwen 2.5 Coder also shows the highest error count for the 'Shared Cars Locations' dataset, with a total of 22 errors. The 'Netflix' dataset for Qwen accumulated 7 errors, the 'COVID-19 Twitter' dataset 6 errors, the 'Madrid Daily Weather' dataset 4 errors, and the 'Supermarket Sales' dataset 2 errors.

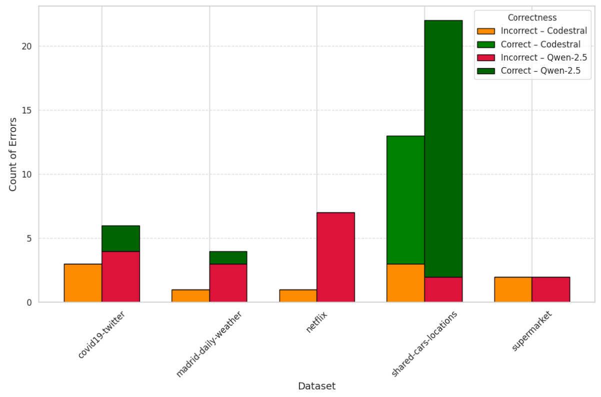

**FIGURE 23.** The amount of errors found in the tests, per dataset.

As for the functional correctness distribution, in most cases, errors in the datasets led to incorrect results, indicating that the code impacted the desired output. However, in the 'Shared Cars Locations' dataset, 10 tests for Codestral and 20 tests for Qwen 2.5 Coder contained errors in the code,

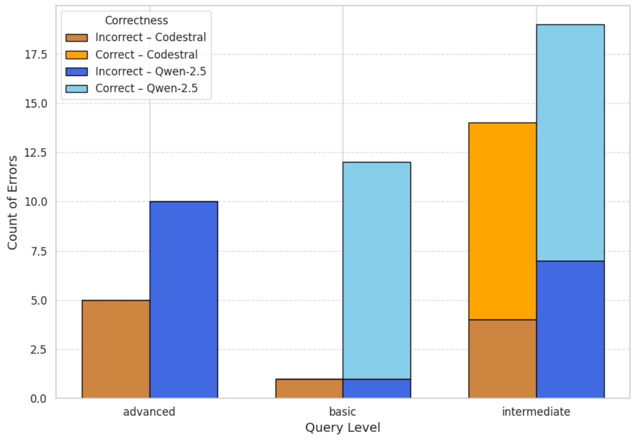

**FIGURE 24.** The total count of errors, grouped by query complexity levels.

but still produced correct results. Most tests were from the dataset's two intermediate queries, as depicted in Fig. 24, although Qwen had a fair amount of such tests (with errors in code but correct output) in basic queries as well. The reason behind this issue is that both LLMs interpret the dataset's timestamps. They fail to modify them as required, but PySpark handles datetime data internally, even if the initial LLMs' conversion or type specification was not perfectly aligned with what was expected by the system. For Codestral, these 10 occasions represent 20% of the total tests for the 'Shared Cars Locations' dataset. In case of Qwen, these 20 occasions represent 40% of the same dataset. An early assumption for this system's behavior could be that both Codestral and Qwen models need further clarification in understanding the context and relationships between columns in location data (e.g., pairing location information with their corresponding timestamps). Additional analysis should be conducted, along with other types of location data for further validation, such as indoor positioning information [83].

## VI. CRITICAL ASSESSMENT

### A. STRENGTHS AND LIMITATIONS

The results showed the model's strong performance in producing code scripts that met the objectives. For Codestral, out of 250 tests, 87% were fully successful, meaning that these tests were functionally correct and had no errors in their code. At the same time, Qwen 2.5 Coder achieved a fully successful rate of 80% over the same number of tests. In addition, both models' tests had code with readability scores between '2' and '3' (with only 2 out of 250 tests having a score of '1' for Qwen), indicating that — in general — the code was easy for a human to interpret. Furthermore, while Codestral had 80% of its tests fully automated, meaning that human intervention for minor code refinements was relatively not necessary, Qwen exhibited an even higher automation rate at 96.5%. Bottom line, regardless of the queries' complexity, Codestral achieved a 91% score in producing correct code (irrespective of automation levels and minor errors), while Qwen attained a 90% score. These results illustrate the robust code-generation capabilities of both models, showing solid signs of success. Despite the potential difficulties of code generation, stemming from the intricacy of requesting PySpark code rather than classic Python commands, each model managed to provide correct code snippets.

Overall, both models demonstrated that it is feasible to conduct data analysis using an offline LLM without loading data to the model, but by generating specialized code and applying it to the data through a well-designed platform. These findings further demonstrate that the proposed approach effectively materializes the initial motivations of this study. The generation of code within a secure, on-premises environment, enables offline LLMs to address key challenges such as data security, integrity, and privacy. This successful demonstration also suggests that offline LLMs could be further validated for generating code in data analytics tasks within organizational contexts, providing a viable alternative to online models that pose data exposure risks, and also potentially replacing existing data analytics frameworks.

However, specific concerns were also identified during the study. Despite the robust hardware specifications of the physical machine hosting the LLM, they were insufficient for quick, real-time code generation. The average time taken to produce a response was about 60 seconds for Codestral, and 25 seconds for Qwen 2.5 Coder. Both averages are not ideal for real-world scenarios, especially Codestral's. Although the GPU and CPU usage rates were moderate, GPU memory utilization was consistently high. A need for a more powerful GPU became evident early on, in order to provide faster responses. This serves as the main limitation for future expansion of this study, based on the current capabilities of offline LLMs.

Large language models are a rapidly evolving field, particularly in recent years. Given the current state of technology, a significant upgrade to a more powerful GPU appears to be the only way to drastically speed up the responses by the LLM. The upgraded GPU should have enough dedicated memory to fully offload the model. In order to achieve near-real-time responses by the model, a highly powerful server would need to be dedicated solely to the model. Given the high requirements of the desired GPU, the cost for such a server would be substantial for any organization. Whether it would be worth investing to such a physical machine, only to take advantage of offline LLMs for on-premise data analysis, is up for each organization to decide. However, the advantage of eliminating data privacy and sensitivity concerns is still a compelling reason to consider the use of offline LLMs.

Apart from the performance shortages, the automation of the process — mainly for Codestral — should also be improved. In 20% of its tests, human intervention was needed for minor code adjustments, such as removing additional text, or specifying the variable to where the final result should be stored, despite the prompt's explicit instructions in both cases. In real-world scenarios, ensuring full automation is essential. Qwen 2.5 Coder failed to produce automation-ready code in only 4.4% of its tests, which highlights its readiness for potential real-world deployment.

Lastly, while both Codestral and Qwen 2.5 Coder demonstrated robust performance by following the commands of the main prompt message, it was observed that Qwen initially had the tendency to partially generate standard Python code, rather than strictly valid PySpark operations. To address this, an additional instruction was appended to the prompt for Qwen: 'Ensure the code runs as valid PySpark DataFrame operations, not standard Python, and verify its execution within Spark.' This adjustment emphasizes the need for model-specific prompt tuning. It does not represent a failure, but rather an iterative enhancement in the evaluation framework.

## B. FUTURE WORK

Focusing on improving the code generation process, which is the essence of this study's subject, future work will prioritize fine-tuning of Codestral and Qwen 2.5 Coder to potentially achieve even better results. If other large language models emerge that rival the selected models in terms of efficiency and performance, they could also be taken under consideration. The goal is to fine-tune Codestral and Qwen to increase the levels of automation, by ensuring it reduces unnecessary text when generating code, and consistently stores final results in a pre-determined variable recognizable by the data processing platform, as was done in the current study. While the levels of error-free tests and correct final results in this study are high, the evaluation of a fine-tuned model should also improve these metrics as well. In summary, a fine-tuned offline model, fully offloaded to a more powerful GPU, could enhance the outcomes of a similar follow-up study, potentially improving the evaluation results in almost all aspects. However, rigorous safeguards must be implemented during fine-tuning to prevent the introduction of bias, ensuring the model's outputs remain robust and generalizable across diverse data domains.

In addition to these enhancements, future work could also explore hybrid approaches of the framework's infrastructure. Hybrid models integrate offline processing with private cloud infrastructures. Such a configuration would leverage scalable computational resources, in order to enable extensive model optimization and rapid iteration cycles. In addition, it would still preserve the inherent data privacy benefits of an offline framework. However, implementing such a hybrid approach does come with cost implications, as it requires investments in both robust offline systems and scalable cloud resources, along with the complexity of maintenance.

Moreover, the current study could benefit by expanding its focus to include the development of a dedicated user interface. As of now, the research framework — depicted in Fig. 1 — is primarily aimed at assessing the underlying capabilities of the models in a controlled environment. For the current study to move towards a production-ready solution, the importance of an intuitive and user-friendly interface is evident. Such an interface should accommodate both non-technical users and experts, particularly for practical applications and real-world deployment. Additionally, future work could address the incorporation of additional functionalities and tests required, in order to make this study a solid solution for enterprise deployment. Leveraging the model-agnostic design of the proposed framework and the extensibility of the underlying architecture could safely lead to a transition from the current research-based approach, into a fully operational tool suitable for enterprise applications.

## VII. CONCLUSION

This study aimed to explore the capabilities of offline LLMs in generating code for data analysis operations. The use of popular, online models like GPT or Gemini could raise security concerns, due to data privacy and sensitivity regulations. In addition, current models may struggle to manage larger volumes of data, and may not always fully comprehend the context of the data they are provided, leading to incorrect results. The usage of offline LLMs could address both issues, enabling organizations to take advantage of a model's capabilities while keeping both the model and their data locally. In this setup, the model receives natural language queries from the user, understands the context, generates data analysis code, and, through a communication pipeline, the code is being forwarded to a data processing platform where it is applied to the data for the final results' extraction.

The system used Mistral AI's Codestral LLM and Alibaba's Qwen 2.5 Coder, due to their efficiency and better performance against other code-oriented models. Moreover, the communication pipeline and data processing platform were implemented atop of Apache's Spark framework, ensuring efficient data handling and management. Five datasets were selected for the testing and evaluation phase. For each dataset, five queries of varying complexity were authored. For each query, 10 tests were conducted. This resulted in a total of 250 tests for each LLM, evaluating Codestral's and Qwen's ability to generate precise code for the provided queries.

The study's findings highlight the considerable promise of offline LLMs for generating accurate and interpretable code, while preserving data security. Both Codestral and Qwen have demonstrated stellar capabilities in producing functionally correct scripts, with high levels of automation and readability. However, the primary bottleneck remains the computational performance, mainly the need for more powerful GPU resources in order to achieve faster, near real-time responses. Looking ahead, targeted enhancements such as careful model fine-tuning, the development of an intuitive user interface, and the exploration of hybrid infrastructures to optimize scalability and privacy, are areas for potential future research. These steps would not only address current performance constraints. They could also guide the transition from a controlled research setting, to an enterprise-ready solution.

This study could lay the groundwork for other future research efforts, aiming to establish offline large language models (LLMs) as a viable alternative for data analysis operations, using code generation based on natural language queries. Instead of bringing the data to the LLM, this approach suggests bringing the LLM's code to the data. The exploration of the large language models' applicability to the field of Data Science will continue. Future applications are likely to fully harness the power of LLMs for optimal data processing and analysis. Over time, this study could be one of the standard methodologies of analyzing data. One thing is certain: Large language models will continue to expand their influence across all scientific fields, with Data Science being a key area of integration. How deeply LLMs will embed themselves in current data analysis trends remains to be seen.

## APPENDIX A
## THE PYTHON CODE READABILITY CALCULATION FUNCTION

```
def evaluate_readability(code):
"""
The function analyzes the code to determine the readability score
        by:
— Checking the length of each command.
— Counting the number of method call chains.
— Analyzing the depth of nested structures.
"""
try:
# Split the code into individual commands based on newlines and
        semicolons
commands = [cmd.strip() for cmd in code.replace('\n', ';').split(';
        ') if cmd.strip()]

readability_score = 0 # Initialize the readability score
total_commands = len(commands) # Get the total number of
        commands

if total_commands == 0:
return readability_score

# Initialize counters for long lines and maximum chain length
long_lines = 0
max_chain_length = 0

for cmd in commands:
# Split the command into lines to check for long lines
lines = re.split(r';\s*|\n', cmd)
long_lines += sum(1 for line in lines if len(line) > 80) # Count
        lines longer than 80 characters

# Parse the command into an abstract syntax tree (AST)
try:
tree = ast.parse(cmd)
except SyntaxError:
continue # Skip this command if there's a syntax error

# Count method calls in chained operations
for node in ast.walk(tree):
 if isinstance(node, ast.Call):
 chain_length = 0
 current_node = node.func
```

**Listing 5.** **A custom function to provide a readability score, from 1 to 3 (with 3 meaning higher readability), to the LLM's generated code.**

```
# Traverse the chain of method calls to count its length
while isinstance(current_node, ast.Attribute):
chain_length += 1
current_node = current_node.value
max_chain_length = max(max_chain_length, chain_length)

# Adjust readability score for long lines
if long_lines / total_commands < 0.1: # Allow some tolerance for
        long lines
readability_score += 1

# Adjust readability score based on the length of method call
        chains

 if max_chain_length <= 3: # Full score for chains up to 3 method
        calls
 readability_score += 1

 # Parse the entire code to evaluate nested structures
 try:
 tree = ast.parse(code)
 except SyntaxError:
 return readability_score # Return the score if there's a syntax
        error in the entire code
def count_nested_structures(node, depth=0):
# Increase depth for specific nodes (if, for, while, function
        definitions)
if isinstance(node, (ast.If, ast.For, ast.While, ast.FunctionDef)):
depth += 1
# Recursively check the depth of nested structures
return max([count_nested_structures(child, depth) for child in
        ast.iter_child_nodes(node)], default=depth)

nested_structure_depth = count_nested_structures(tree)
# Full score if the maximum depth of nested structures is 2 or less
if nested_structure_depth <= 2:
readability_score += 1

return readability_score # Return the final readability score
except Exception as e:
print(f"Errorevaluatingreadability:{e}")
return 0 # Return a score of 0 if there's an error
)
```

**Listing 5.** ***(Continued.)*** **A custom function to provide a readability score, from 1 to 3 (with 3 meaning higher readability), to the LLM's generated code.**

## APPENDIX B SUMMARY OF THE 'SHARED CARS LOCATIONS' DATASET

```
{
    "intro": "Thisdatasetcontainsinformationaboutsharedcarlocations,
        includingvariousattributesthatdescribeeachentry.",
    "columns": [
        {
            "name": "latitude",
            "type": "float64",
            "simplified_type": "float",
            "description": "Latitudecoordinateofthecar'slocation.",
            "sample_values": [
                32.083,
                32.064615,
                32.113535
            ]
        },
```

**Listing 6.** **A dataset summary of the "Shared Cars Locations" dataset, carefully crafted by a research team member.**

```
{
    "name": "longitude",
    "type": "float64",
    "simplified_type": "float",
    "description": "Longitudecoordinateofthecar'slocation.",
    "sample_values": [
        34.8043,
        34.77577,
        34.7911
    ]
},
{
    "name": "total_cars",
    "type": "int64",
    "simplified_type": "int",
    "description": "Totalnumberofcarsavailableatthelocation.",
    "sample_values": [
        0,
        1,
        3
    ]
},

 {
    "name": "cars_list",
    "type": "object",
    "simplified_type": "string",
    "description": "Listofcaridentifiersavailableatthelocation.",
    "sample_values": [
        "[]",
        "[213]",
        "[137,180,193]"
        ]
    },
    {
        "name": "timestamp",
        "type": "object",
        "simplified_type": "datetime",
        "description": "TimestampofthedatarecordinUTC.",
        "sample_values": [
            "2019—12—0621:51:02UTC",
            "2019—11—2914:00:02UTC",
            "2019—09—2013:21:03UTC"
        ],
        "datetime_format": "%Y—%m—%d%H:%M:%S%Z"
    }
],
"note": "Thesampledataprovidedhereissyntheticandintendedtoillustrate
        thedatastructure."
}
```

**Listing 6.** ***(Continued.)*** **A dataset summary of the "Shared Cars Locations" dataset, carefully crafted by a research team member.**

## ACKNOWLEDGMENT

The data generated and analyzed during the current work are available from the corresponding author upon reasonable request. Similarly, the code used in the current work can be provided by the corresponding author upon reasonable request.

## REFERENCES

[1] *Gpt-4o Large Language Model*, OpenAI, San Francisco, CA, USA, 2024.

[2] *Gemini Large Language Model*, Google, Mountain View, CA, USA, 2024.

[3] H. Naveed, A. Ullah Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," 2023, *arXiv:2307.06435*.

[4] W. Xin Zhao et al., "A survey of large language models," 2023, *arXiv:2303.18223*.

[5] M. Research AI4Science and M. Azure Quantum, "The impact of large language models on scientific discovery: A preliminary study using GPT-4," 2023, *arXiv:2311.07361*.

[6] A. Nikolakopoulos, S. Evangelatos, E. Veroni, K. Chasapas, N. Gousetis, A. Apostolaras, C. D. Nikolopoulos, and T. Korakis, "Large language models in modern forensic investigations: Harnessing the power of generative artificial intelligence in crime resolution and suspect identification," in *Proc. 5th Int. Conf. Electron. Eng., Inf. Technol. Educ. (EEITE)*, May 2024, pp. 1–5.

[7] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, "Challenges and applications of large language models," 2023, *arXiv:2307.10169*.

[8] E. Kasneci et al., "ChatGPT for good? On opportunities and challenges of large language models for education," *Learn. Individual Differences*, vol. 103, Mar. 2023, Art. no. 102274.

[9] A. Bewersdorff, K. Seßler, A. Baur, E. Kasneci, and C. Nerdel, "Assessing Student errors in experimentation using artificial intelligence and large language models: A comparative study with human raters," *Comput. Educ., Artif. Intell.*, vol. 5, Jan. 2023, Art. no. 100177.

[10] C. Ebert and P. Louridas, "Generative AI for software practitioners," *IEEE Softw.*, vol. 40, no. 4, pp. 30–38, Jul. 2023.

[11] U. Khandelwal, O. Levy, D. Jurafsky, L. Zettlemoyer, and M. Lewis, "Generalization through memorization: Nearest neighbor language models," 2019, *arXiv:1911.00172*.

[12] R. H. Hariri, E. M. Fredericks, and K. M. Bowers, "Uncertainty in big data analytics: Survey," *J. Big data*, vol. 6, no. 1, pp. 1–16, 2019.

[13] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[14] F. M. Suchanek and L. A. Tuan, "Knowledge bases and language models: Complementing forces," in *Proc. Int. Joint Conf. Rules Reasoning*, Jan. 2023, pp. 3–15.

[15] K. Kang, Y. Yang, Y. Wu, and R. Luo, "Integrating large language models in bioinformatics education for medical students: Opportunities and challenges," *Ann. Biomed. Eng.*, vol. 52, no. 9, pp. 2311–2315, Sep. 2024.

[16] *General Data Protection Regulation*, EU law, Eur. Commission, Brussels, Belgium, 2016.

[17] Q. Wang, M. Du, X. Chen, Y. Chen, P. Zhou, X. Chen, and X. Huang, "Privacy-preserving collaborative model learning: The case of word vector training," *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 12, pp. 2381–2393, Dec. 2018.

[18] Y. Feng, S. Vanam, M. Cherukupally, W. Zheng, M. Qiu, and H. Chen, "Investigating code generation performance of ChatGPT with crowdsourcing social data," in *Proc. IEEE 47th Annu. Comput., Softw., Appl. Conf. (COMPSAC)*, Jun. 2023, pp. 876–885.

[19] Q. Gu, "LLM-based code generation method for golang compiler testing," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2023, pp. 2201–2203.

[20] S. I. Ross, F. Martinez, S. Houde, M. M´uller, and J. D. Weisz, "The programmer's assistant: Conversational interaction with a large language model for software development," in *Proc. 28th Int. Conf. Intell. User Interfaces*, Mar. 2023, pp. 491–514.

[21] A. Soliman, S. Shaheen, and M. Hadhoud, "Leveraging pre-trained language models for code generation," *Complex Intell. Syst.*, vol. 10, no. 3, pp. 3955–3980, Jun. 2024.

[22] *Conala: The Code/Natural Language Challenge*, CoNaLa, Nat. Sci. Found., Alexandria, VA, USA, 2024.

[23] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*. Lincoln, NE, USA: IEEE Computer Society, Nov. 2015, pp. 574–584, doi: 10.1109/ASE.2015.36.

[24] G. Pinna, D. Ravalico, L. Rovito, L. Manzoni, and A. D. Lorenzo, "Enhancing large language models-based code generation by leveraging genetic improvement," in *Proc. Eur. Conf. Genetic Program. (Part EvoStar)*, Jan. 2024, pp. 108–124.

[25] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "CoderEval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, Feb. 2024, pp. 1–12.

[26] S. Omari, K. Basnet, and M. Wardat, "Investigating large language models capabilities for automatic code repair in Python," *Cluster Comput.*, vol. 27, no. 8, pp. 10717–10731, Nov. 2024.

[27] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *Proc. IEEE/ACM Int. Conf. Softw. Eng., Future Softw. Eng. (ICSE-FoSE)*, May 2023, pp. 31–53.

[28] M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, "Natural language generation and understanding of big code for AI-assisted programming: A review," *Entropy*, vol. 25, no. 6, p. 888, Jun. 2023.

[29] *Github Copilot Ai Developer Tool*, GitHub, San Francisco, CA, USA, 2024.

[30] *Deepmind Alphacode Ai Developer Tool*, Google, Mountain View, CA, USA, 2024.

[31] J. Wang and Y. Chen, "A review on code generation with LLMs: Application and evaluation," in *Proc. IEEE Int. Conf. Med. Artif. Intell. (MedAI)*, Nov. 2023, pp. 284–289.

[32] H.-C. Liu, C.-T. Tsai, and M.-Y. Day, "A pilot study on ai-assisted code generation with large language models for software engineering," in *Proc. Int. Conf. Technol. Appl. Artif. Intell.* Cham, Switzerland: Springer, 2024, pp. 162–175.

[33] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A large language model for verilog code generation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 29, no. 3, pp. 1–31, May 2024.

[34] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification," in *Proc. ACM Softw. Eng.*, Jul. 2024, vol. 1, no. FSE, pp. 2332–2354.

[35] D. Rau and J. Kamps, "Query generation using large language models," in *Advances in Information Retrieval*. Cham, Swizerland: Springer, 2024, pp. 226–239.

[36] D. Sachan, M. Lewis, M. Joshi, A. Aghajanyan, W.-T. Yih, J. Pineau, and L. Zettlemoyer, "Improving passage retrieval with zero-shot question generation," in *Proc. Conf. Empirical Methods Natural Language Process.*, 2022, pp. 3781–3797.

[37] N. Thakur, N. Reimers, A. Rücklé, A. Srivastava, and I. Gurevych, "Ubiquitous knowledge processing lab (UKP-TUDA)," Dept. Comput. Sci., Technische Universität Darmstadt, Darmstadt, Germnay, 2024.

[38] *Trec 2020 Deep Learning Track*, Microsoft, Redmond, WA, USA, 2020.

[39] X. Qu, Y. Wang, Z. Li, and J. Gao, "Graph-enhanced prompt learning for personalized review generation," *Data Sci. Eng.*, vol. 9, no. 3, pp. 309–324, Sep. 2024.

[40] X. Zhou, Z. Sun, and G. Li, "DB-GPT: Large language model meets database," *Data Sci. Eng.*, vol. 9, no. 1, pp. 102–111, Mar. 2024.

[41] *Openai Codex*, OpenAI, San Francisco, CA, USA, 2024.

[42] *Datarobot AI Solutions*, DataRobot, Boston, MA, USA, 2024.

[43] *Thoughtspot AI-Powered Analytics*, ThoughtSpot, Mountain View, CA, USA, 2024.

[44] *Tableau Data Visualization*, Tableau, Seattle, WA, USA, 2024.

[45] *Microsoft Power BI*, Microsoft, Redmond, WA, USA, 2024.

[46] *What is Prompt Engineering?*, IBM, Armonk, NY, USA, 2024.

[47] G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende, "Prompt engineering in large language models," in *Proc. Int. Conf. Data Intell. Cognit. Informat.*, Cham, Switzerland: Springer, Jan. 2024, pp. 387–402.

[48] J. D. Velásquez-Henao, C. J. Franco-Cardona, and L. Cadavid-Higuita, "Prompt engineering: A methodology for optimizing interactions with AI-language models in the field of engineering," *DYNA*, vol. 90, no. 230, pp. 9–17, Nov. 2023.

[49] W. Cain, "Prompting change: Exploring prompt engineering in large language model AI and its potential to transform education," *TechTrends*, vol. 68, no. 1, pp. 47–57, Jan. 2024.

[50] P. Sahoo, A. Kumar Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," 2024, *arXiv:2402.07927*.

[51] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models," in *Proc. ACM Program. Lang.*, vol. 7, Jun. 2023, pp. 1946–1969.

[52] I. Arawjo, C. Swoopes, P. Vaithilingam, M. Wattenberg, and E. L. Glassman, "ChainForge: A visual toolkit for prompt engineering and LLM hypothesis testing," in *Proc. CHI Conf. Human Factors Comput. Syst.*, May 2024, pp. 1–18.

[53] A. Nikolakopoulos, E. Chondrogiannis, E. Karanastasis, M. J. L. Osa, J. A. Aroca, M. Kefalogiannis, V. Apostolopoulou, E. Deligeorgi, V. Siopidis, and T. Varvarigou, "Scalable data profiling for quality analytics extraction," in *Proc. IFIP Int. Conf. Artif. Intell. Appl. Innov.*, Cham, Switzerland: Springer, 2024, pp. 177–189.
[54] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Proc. AAAI Conf. Artif. Intell.*, vol. 32, Apr. 2018, pp. 1–11.
[55] *NETFLIX: Movies and Tv Shows Dataset*, Kaggle, San Francisco, CA, USA, 2024.
[56] *COVID-19 Twitter Dataset*, Kaggle, San Francisco, CA, USA, 2024.
[57] *Shard Cars Locations Dataset: Location History of Shared Cars*, Kaggle, San Francisco, CA, USA, 2024.
[58] *Autotel Project in Tel Aviv*, AutoTel, Tel Aviv-Yafo, Israel, 2024.
[59] *Madrid Daily Weather Dataset: Daily Weather Conditions in Madrid from 1997–2015*, MavenAnalytics, Kaggle, Mountain View, CA, USA, 2024.
[60] *Supermarket Sales Dataset: Historical Record of Sales Data in 3 Different Supermarkets*, Kaggle, San Francisco, CA, USA, 2024.
[61] V. Dogra, S. Verma, Kavita, M. Woźniak, J. Shafi, and M. F. Ijaz, "Shortcut learning explanations for deep natural language processing: A survey on dataset biases," *IEEE Access*, vol. 12, pp. 26183–26195, 2024.
[62] *Codestral Large Language Model*, MistralAI, Paris, France, 2024.
[63] *Qwen 2.5 Coder Large Language Model*, A. Qwen, Hangzhou, China, 2024.
[64] *Humaneval Dataset*, OpenAI, San Francisco, CA, USA, 2024.
[65] T. Liu, C. Xu, and J. McAuley, "RepoBench: Benchmarking repository-level code auto-completion systems," 2023, *arXiv:2306.03091*.
[66] *'Cruxeval: Code Reasoning,' Understanding, and Execution Evaluation*, Meta, Menlo Park, CA, USA, 2024.
[67] *Codellama: A State-of-the-Art Large Language Model for Coding*, Meta, Menlo Park, CA, USA, 2023.
[68] *Deepseek Coder 33b Large Language Model*, DeepSeekAI, Zhejiang , China, 2024.
[69] *Llama 3: Openly Available Large Language Model*, Meta, Menlo Park, CA, USA, 2024.
[70] *What is Mistral's Codestral? Key Features, Use Cases, and Limitations*, DataCamp, New York, NY, USA, 2024.
[71] B. Hui et al., "Qwen2.5-coder technical report," 2024, *arXiv:2409.12186*.
[72] S. Zhu, W. Hu, Z. Yang, J. Yan, and F. Zhang, "Qwen-2.5 outperforms other large language models in the Chinese national nursing licensing examination: Retrospective cross-sectional comparative study," *JMIR Med. Informat.*, vol. 13, Jan. 2025, Art. no. e63731.
[73] *Apache Spark: Unified Engine for Large-Scale Data Analytics*, Apache, Apache Softw. Found., Wilmington, DE, USA, 2024.
[74] Q. Yin, J. Wang, S. Du, J. Leng, J. Li, Y. Hong, F. Zhang, Y. Chai, X. Zhang, X. Zhao, M. Li, S. Xiao, and W. Lu, "An adaptive elastic multi-model big data analysis and information extraction system," *Data Sci. Eng.*, vol. 7, no. 4, pp. 328–338, Dec. 2022.
[75] A. Nikolakopoulos, M. Julian Segui, A. B. Pellicer, M. Kefalogiannis, C.-A. Gizelis, A. Marinakis, K. Nestorakis, and T. Varvarigou, "BigDaM: Efficient big data management and interoperability middleware for seaports as critical infrastructures," *Computers*, vol. 12, no. 11, p. 218, Oct. 2023.
[76] E. Yom-Tov, S. Fine, D. Carmel, and A. Darlow, "Learning to estimate query difficulty: Including applications to missing content detection and distributed information retrieval," in *Proc. 28th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retr.*, Aug. 2005, pp. 512–519.
[77] J. Guo and Y. Lan, "Query classification," in *Query Understanding for Search Engines*. Springer, 2020, pp. 15–41.
[78] Y. Belinkov and J. Glass, "Analysis methods in neural language processing: A survey," in *Proc. Trans. Assoc. Comput. Linguistics*, vol. 7, Apr. 2019, pp. 49–72.
[79] E. M. Bender and A. Koller, "Climbing towards nlu: On meaning," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 5185–5198.
[80] *Hugging Face: Codestral V01 Large Language Model*, HuggingFace, New York, NY, USA, 2024.
[81] *Lm Studio Server*, LMStudio, New York, NY, USA, 2025.
[82] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," 2019, *arXiv:1904.09751*.
[83] A. Nikolakopoulos, A. Psychas, A. Litke, and T. Varvarigou, "Leveraging indoor localization data: The transactional area network (TAN)," *Electronics*, vol. 13, no. 13, p. 2454, Jun. 2024.

**ANASTASIOS NIKOLAKOPOULOS** received the Diploma degree in information technology from the Harokopio University of Athens, in 2018, and the M.Sc. degree in machine learning and data science from the National Technical University of Athens (NTUA), in 2021, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. He is also a Research Engineer with the Telecommunications Laboratory, NTUA. With extensive involvement in several EU-funded research projects, his expertise spans big data management, advanced data analysis, data space ecosystems, and large language model (LLM) engineering.

**ANTONIOS LITKE** received the Diploma degree from the Department of Computer Engineering and Informatics, University of Patras, Greece, in 1999, and the Ph.D. degree from the Electrical and Computer Engineering Department, National Technical University of Athens (NTUA), in 2006. He was an Adjunct Assistant Professor with the Computer Science Department, University of Ioannina. He is currently a Lecturer with the Hellenic Army Academy, as well as a Researcher with the Institute of Communication and Computer Systems (ICCS), NTUA. He is the author of more than 30 scientific articles (with more than 600 citations) and a reviewer of several international journals and conferences. His research interests include cloud computing, parallel and distributed computing, service-oriented architectures, and information and knowledge engineering.

**ALEXANDROS PSYCHAS** was born in May 1987. He received the Diploma degree in digital systems and the M.Sc. degree in electronic education from the University of Piraeus, in 2011 and 2014, respectively, and the Ph.D. degree from the Department of Electrical and Computer Engineering, NTUA. He is currently a Senior Research Engineer and a Software Engineer with the Telecommunications Laboratory, and also with the Institute of Communication and Computer Systems (ICCS), NTUA, where he has participated in numerous EU projects.

**ELENI VERONI** received the Diploma degree in digital systems and the M.Sc. degree in digital systems security from the University of Piraeus, Greece, in 2013 and 2015, respectively. Currently, she is an R&I Project Manager with Netcompany-Intrasoft S.A., contributing to several initiatives aimed at enhancing civil security for society. Since 2015, she has been actively engaged in numerous EU-funded research and innovation projects. Her research interests include personal identification systems, biometrics, data security, and privacy.

**THEODORA VARVARIGOU** received the B.Tech. degree from the National Technical University of Athens, in 1988, the M.S. degree in electrical engineering and the M.S. degree in computer science from Stanford University, Stanford, CA, USA, in 1989 and 1991, respectively, and the Ph.D. degree from Stanford University, in 1991. She was with the AT&T Bell Laboratories, Holmdel, NJ, USA, from 1991 to 1995. From 1995 to 1997, she was an Assistant Professor with the Technical University of Crete, Chania, Greece. Since 1997, she has been an Assistant Professor and a Professor with the National Technical University of Athens, since 2007. She has great experience in the area of semantic web technologies, scheduling over distributed platforms, embedded systems, and grid computing.