Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Domain-Aware Prompting with LLMs for Data Science Notebooks

Author:
Sanjeet Chatterjee

Supervisor:
Dr. Pedro Baiz

Second Marker:
Dr. Sergio Maffeis

August 31, 2024

Abstract

LLMs have shown impressive capabilities across a variety of tasks, such as semantic understanding, reasoning capabilities, and code synthesis. This has the potential to assist data scientists in data preparation and exploration tasks. Computational notebooks have become popular tools for data scientists to explore, analyse and prepare data. Notebooks provide a richer contexual environment in comparison to single-step code generation tasks and benchmarks due to its live execution state and multi-step nature.

We build upon the ARCADE benchmark and experiments [1] and propose novel prompting techniques that leverages contextual information in a stateful notebook environment. We explore combining execution state with various prompting methods to improve on the original ARCADE benchmark results and evaluate our prompting methods with the Llama 3 family of models. Our experiments highlight the importance designing domain-specific prompting that takes into context all available information to enable stronger model performance on data science tasks in a stateful notebook environment.

Contents

Chapter 1

Introduction

1.1 Motivation

Interactive environments such as Jupyter notebooks have become popular tools for data scientists to explore, analyse and prepare data. Notebooks consist of markdown and code cells and execution state is preserved across code cell runs. These environments allow for rapid prototyping and iteration, but such work can often be repetitive and time-consuming. This has led to developments in automation and assistive tools to accelerate data science workflows. Various statistical and machine-learning methods exist to address and automate these problems separately however, they can require significant training efforts upfront or require strong domain knowledge.

Recent developments in Large Language Models (LLMs) demonstrate impressive capabilities across a variety of tasks, simply through in-context learning with a few provided examples. LLMs demonstrate semantic understanding, reasoning capabilities and code synthesis which have the potential to assist with steps data scientists go through for data preparation and exploration. There are many benchmarks for code generation tasks across various domains and programming languages, however, these are constrained to a single question and answer format. Notebooks offer additional challenges due to the multi-step nature of notebooks, reflecting real-world data science workflows.

JuICe [2] is a refined curated corpus of over a million Jupyter notebooks, however these are not executable and therefore evaluation is restricted to surface-form metrics rather than functional accuracy. Derivations of JuICe attempt to address this to allow for evaluation by execution but do not always reflect real-world data science workflows [3, 4]. ARCADE [5] is a similar benchmark consisting of 133 notebooks. It consists of existing and crafted notebooks and is designed to reflect real-world natural language intents and data science problems, written by professional data scientists. The ARCADE paper also evaluates the PaLM model [6], as well as PaChiNCo - PaLM fine-tuned on a dataset of notebooks. As stated in the paper, they only utilise simple schema descriptions from the notebooks and prompting techniques are limited.

We improve on the original benchmarks by using all available information in the execution state, including variables and cell outputs. We propose novel extraction and serialization methods to generate a more comprehensive prompt, which better reflects the notebook state and the information available to data scientists working on such notebooks.

1.2 Contributions

In this work, we utilise a variety of techniques to extract and serialize a notebook's execution state and generate prompts based on the ARCADE dataset.

- **Execution Metadata** We run every notebook and iteratively run each code cell to extract the execution state of each notebooks. This is cached across runs and is used to generate prompts.
- **DataFrame Serialization** DataFrames and Series are foundational to many data science workflows. For prompts, the objective is to serialize such variables in a way that captures all contextual information whilst limiting prompt size.
- **DataFrame Dependency Graph** In a notebook, DataFrame variables undergo a series of transformations and operations. This results in a dependency graph which follows and branches out as

operations are performed and intermediate variables are created, often starting from DataFrame read from disk. Through AST parsing, we extract and generate a dependency graph that enables the smarter prioritisation of variables included in the prompt, as well as better visibility into the dataset.

- **Code Metadata Extraction** AST parsing also allows the extraction of additional metadata, such as imports, function calls and user-defined functions which can be used for both prompts and analysis.
- **Prompting Techniques** We explore simple vanilla prompting similar to the original paper. This is followed by more complex techniques such as Chain-Of-Thought (CoT) [23] and multi-step prompting.

Chapter 2

Background

2.1 Large Language Models

Large Language Models (LLMs) typically refer to language models based on the transformer architecture [7]. It introduces the attention mechanism which captures long range dependencies in the input sequence. Recent LLMs use decoder-only transformers, which are autoregressive models that generate the output sequence one token at a time, given the previous tokens in the sequence.

Scaling such models to billions of parameters [8] has led to emergent capabilities such as in-context learning, multi-step reasoning and instruction following [9]. These models achieve surprisingly good zero-shot performance on domains outside of their training data [6] and can match state-of-the-art performance of smaller specialized models for certain tasks, by leveraging few-shot prompting [10]. Similar to humans, given simple instructions and a few examples, such models can perform complex language tasks without the need for additional fine-tuning on large task-specific datasets.

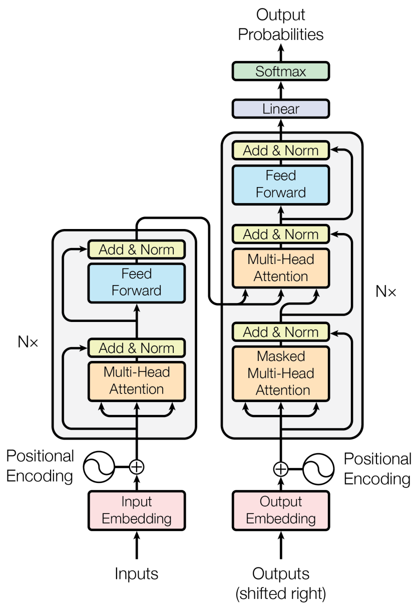


Figure 2.1: Transformer Model Architecture from "Attention Is All You Need" [7]

Initial model pre-training is done on large unlabelled datasets in a variety of domains, through self-supervised learning. At this stage, prompt design can lead to impressive generalised results due to in-context learning [10]. However, this is limited by the context length and may not be suitable for complex domain-specific tasks. Instruction tuning leads to substantial zero-shot performance improvements over even few-shot performance of foundation models [11]. This requires the curation of valid instruction-response demonstrations which can be expensive.

2.2 Natural Language to Code

LLMs have also demonstrated impressive results in program synthesis, translating natural language specifications to code [12]. However, there is no guarantee of correctness or quality of generated code since these models do not recognise the syntax or semantics of the programming language. Instead, these models have learnt the underlying grammar, how various libraries work and what human-readable code looks like, having been pre-trained on massive datasets of code [13]. Additional steps are required to ensure the robustness of code synthesis.

One approach is utilising in-context learning to add additional prompt context to guide the model towards the correct solution. This can be in the form of providing natural language human feedback [14], including context-relevant example pairs [15], and directly incorporating execution traces for unit tests [16]. LLMs have also shown potential in multi-step reasoning and generating feedback which can complement other few-shot techniques to critique and improve its final output [17].

Another observation is that the correct/desired solution is likely to be generated amongst multiple samples by the same model [18]. The simplest method is selecting the sample with the highest average log likelihood of the output tokens with length normalization applied [18]. Samples with syntax and semantic errors can be discarded, followed by grouping and ranking to get top k outputs, all based on execution results [19]. This can work on data tasks with samples tested on a small subset of the dataset, for which a similarity metric can be computed between the execution output and the desired result [20]. However, it is important to note that generating multiple samples can be computationally expensive and adds additional latency. Temperature mixing can be used to increase the diversity of samples generated, and increase the chance of finding the correct solution [19]. The temperature parameter controls the randomness of selecting the next token, with higher temperatures resulting in more creative outputs.

Various sampling methods have been proposed, which is the strategy to choose the next token given the log probabilities of all possible tokens. The simplest approach is greedy search which chooses the highest probability token at each step. Beam search keeps track of the top k tokens at each step, and chooses the sequence with the highest probability at the end. Top-k sampling only considers the top k tokens at each step, and randomly samples from them, with nucleus sampling varying the number of tokens based on the token probability distribution [21]. The intuition for this is that the model may be more confident on a small set of tokens at a certain step, such as a return statement when generation code. Masked decoding constrains the output token space at each step, for when the output grammar is known. This can be coupled with autocomplete systems, tailored to the specific task, for example restricting possible variable names to avoid naming errors [22].

Code repair as a post-processing step is another approach. In many cases, samples capture the correct intent but contain simple syntax or semantic errors. Input context-based methods can be used for repair in post, such as variable reference or argument errors, by traversing the AST [15].

2.3 Prompting Methods

The effectiveness of in-context learning [10] has given rise to various prompting techniques to guide the model towards the correct solution. Chain-of-thought (CoT) breaks down tasks into multi-step problems to explore reasoning and planning capabilities, with the few-shot examples demonstrating the step-by-step approach to solving the problem [23]. Hand-crafted demonstrations can be time-consuming so the zero-shot CoT approach sees surprising improvements on various benchmarks by just adding "Let's think step by step" to the prompt [24]. Auto-CoT improves on this by generating the step-by-step demonstrations for problems with zero-shot CoT and inserting examples which lead to the correct solution into the prompt at test time [25]. Self-consistency replaces greedy decoding with sampling a diverse set of reasoning paths, followed by aggregating and selecting the most consistent result [26]. ReAct (Reasoning and Acting) interleaves reasoning, action and observation steps which makes it less prone to hallucination than CoT but can increasing reasoning error rate [27]. ReACT and CoT-SC can be combined and switched if found to not be converging, to achieve high performance on various benchmarks [27]. Structured CoT boosts CoT for code generation by outputting pseudocode as an intermediate step, with loop and branch structures instead of just sequential steps [28]. Tree-of-Thought (ToT) maintains a tree structure of the thoughts and combines this with search algorithm to enable exploration and backtracking in a systematic approach [29]. Figure 2.2 shows a visual comparison between some the key prompting methods.

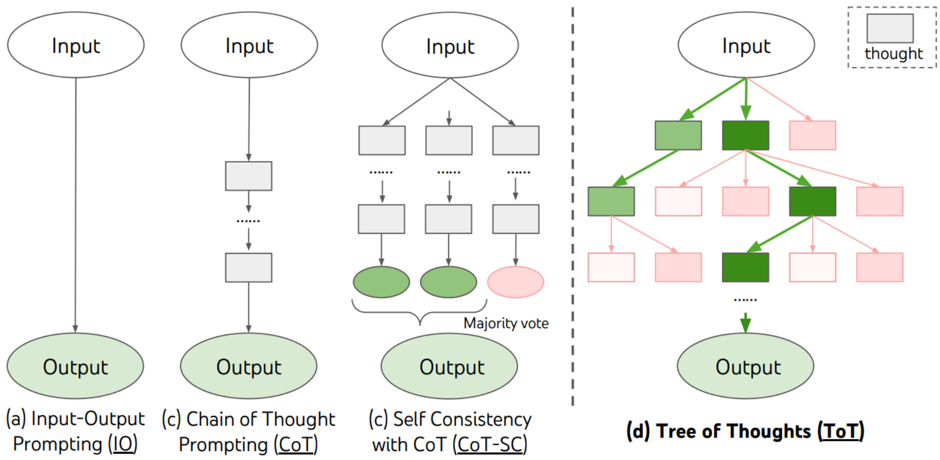


Figure 2.2: Prompting Methods from "Tree of Thoughts" [29]

Selecting context-relevant examples is itself an important task. With a bank of example pairs relevant to the task at hand, similarity metrics can be computed on the user input with sentence embeddings [30]. This can be grown overtime with human feedback to increase the relevance of examples included over time [15].

2.4 Data Wrangling

Data cleaning, matching and transformation are essential steps in data preparation for downstream tasks such as analytics or machine learning. This is often a manual, time-consuming process and much work has been done to assist and automate this process. These are often multi-step pipelines to transform the data into the desired schemas with correct formats.

Programming by example (PBE) is the process of synthesising programs from user-provided input-output examples. Foofah leverages PBE and searches the space of possible data transformations with the classic A* path-finding algorithm with an effective heuristic function [31]. Transform-Data-by-Example (TDE) shows significant improvement by indexing a large variety of transformation functions crawled from GitHub and Stack Overflow [32]. Static analysis, code restructuring and dynamic execution-based analysis paired with generated representative examples is used to index functions for retrieval during program synthesis. Auto-Pipeline builds multi-step pipelines with search and reinforcement learning based synthesis algorithms and replaces PBE with a target based approach [33].

Applying LLMs to classic data tasks is an interesting area of exploration due to being adaptable to various downstream tasks with in-context learning. GPT 3.5 can outperform most previous SOTA methods in entity matching, data cleaning and data integration with relatively simple prompting [34]. However, this runs each data record through a prompt which is not always feasible in real-world use cases due to time, scale and cost constraints.

Serializing tabular data in LLMs is a challenge since such models are autoregressive and operate on sequential data. TabLLM explores different methods of serializing records for working with tabular data with few-shot prompting to answer simple classification tasks [20]. ReAcTable leverages LLM code generation for table question answering, with external Python/SQL code executors manipulating intermediate data representations in order to answer the question using a more accessible format for the LLM. This is achieved using the ReAct framework [27]. Figure 2.3 showcases this approach. Voting mechanisms with multiple samples are used to facilitate exploration to address ambiguity and improve reasoning abilities.

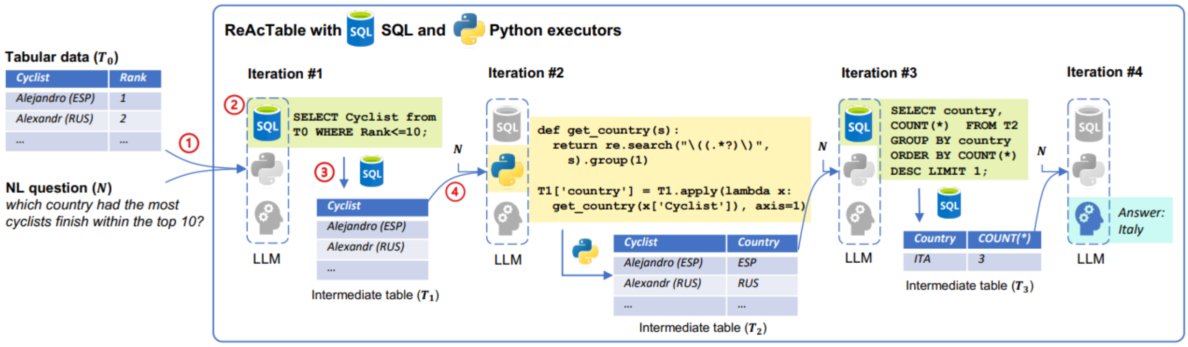


Figure 2.3: ReAcTable framework from "ReAcTable" [35]

2.5 LLMs for Computational Notebooks

Many benchmarks such as HumanEval [18], MBPP [13] and LiveCodeBench [36] exist for Python code generation tasks, and are commonly used to benchmark LLM models and agents. For data science, DS-1000 contains Python tasks utilising data science libraries such as Pandas, Numpy and Scikit-learn [37]. Rewrite, surface and semantic perturbations are used to generate additional problems, and outputs are evaluated with unit test cases and surface-form metrics. These benchmarks are restricted to a single question / answer format and do not reflect the multi-step nature of data science workflows in notebooks.

Interactive Jupyter notebooks interleaving code and natural language markdown cells have become the standard for data science workflows by enabling faster prototyping and collaboration. Notebooks provide a stateful Python environment which allows for stepped execution of code cells and allows for the preservation of intermediate results. This introduces additional challenges such as tracking DataFrame transformations and schema changes, long-range semantic relationships across notebook cells, ambiguous or under-specified natural language intents in markdown cells and accounting for the stateful nature of the notebook environment.

Table 2.1: Existing Datasets for Data Science Notebooks

Dataset	No. Notebooks	No. Tasks	Tasks / Notebook	Evaluation Method
JuICE	1457	3946	2.7	Exact Match + BLEU
MS DSP	305	1096	3.6	Unit Tests
ExeDS	277	534	1.9	Output Match
ARCADE	133	1082	8.55	Output Match

JuICE contains 1457 notebooks with a total 3946 tasks [2]. These are sourced from GitHub and consist of mostly tutorials, assignments and course notebooks that have detailed instructions and problem definitions that do not necessarily reflect real-world data science tasks. These notebooks are not executable and surface-level metrics is used for evaluation with exact match or BLEU. *DSP* is a subset of the *JuICE* dataset that are executable and are combined with unit tests for automatic evaluation [4]. *ExeDS* is another subset of the *JuICE* dataset and uses hand-annotated intents and compares execution outputs between the generated and reference outputs rather than unit tests [3].

ARCADE consists of 133 notebooks with 1082 tasks, along with corresponding CSV dataset files [5]. It is split into two distinct categories of Existing Tasks and New Tasks.

Existing Tasks Initial notebooks were sourced from the JuICE dataset and BigQuery public datasets were filtered and deduplicated. Candidate cells for annotation were chosen based on preceding markdown cells with an intent or through parsing the AST for pandas function calls. This allowed identification of complex data wrangling tasks and simpler dataset exploration tasks. Target notebooks were grouped by the underlying ML dataset being used and longer notebooks were selected for annotation. Natural language intents were annotated by professional data scientists with the goal of keeping annotations simple, concise and potentially ambiguous to reflect real-world tasks.

New Tasks Tabular datasets were sourced from Kaggle and given to annotators to create new tasks that reflect real-world data analysis and exploration. These were specified to be more challenging with more Pandas API calls on average. Natural language intents were annotated in a similar style as before and notebooks were peer-reviewed for quality assurance.

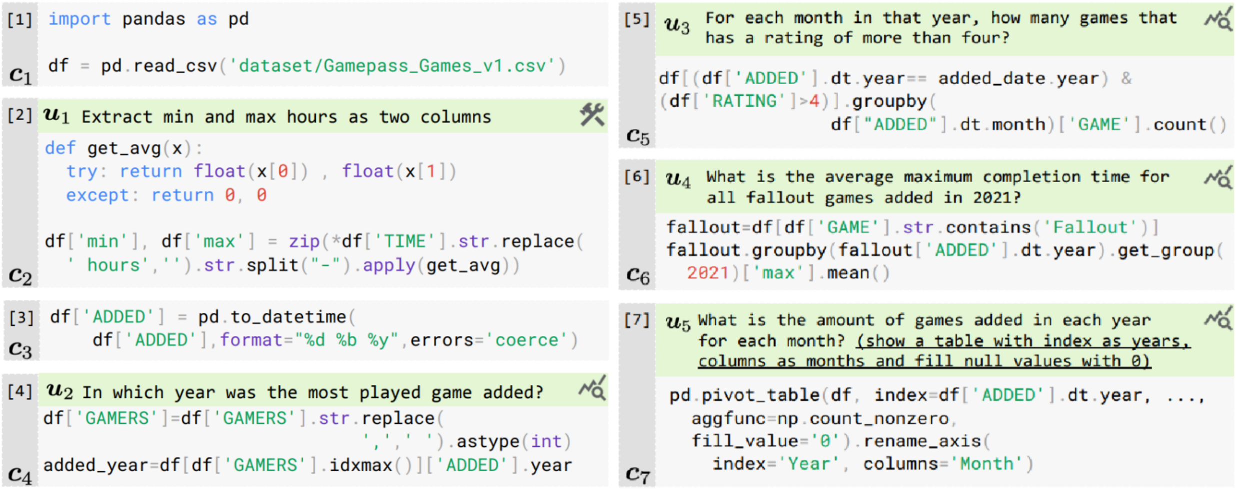


Figure 2.4: Sample ARCADE computational notebook. [1]

Similar to *ExeDS*, *ARCADE* uses execution-based evaluation to evaluate functional correctness. Surface-level metrics like BLEU do not account for the large space of functionally correct programs that match with the reference solution so execution-based evaluation is the right direction. However, *ARCADE* has more complex tasks and dependencies between tasks that combined with simple, concise natural language intents, makes it a more challenging benchmark. Problems require a deeper semantic understanding of the user intents and mapping them to existing variables and the wider rich context of the notebook. Therefore, we chose to build upon the ARCADE benchmark for our experiments.

2.6 ARCADE: Exploratory Analysis

ARCADE consists of 133 notebooks with 1082 tasks. Task output types are shown in Table 2.3 with DataFrame / Series and Numeric outputs making up 82.7% of the dataset. This is expected in the context of data exploration and analysis tasks of the ARCADE notebooks.

Table 2.2: ARCADE Dataset Split

Dataset Split	No. Tasks	No. Notebooks	Tasks / Notebook
Existing Tasks	476	62	7.68
New Tasks	661	70	9.44

Table 2.3: ARCADE Task Output Types

Output Type	%
DataFrame / Series	61.3
Numeric	21.46
String	7.95
List / Tuple / Set	4.41
Empty	3.83
Boolean	0.77
Plot	0.29

The dataset is split into Existing Tasks and New Tasks, with the latter having been created from scratch with the requirement of having more complexity. This means simple ambiguous intents that require disambiguation by using the previous notebook context, deeper semantic understanding of the task at hand and more Pandas API calls on average. For example, Listing 2.1 requires multiple steps of understanding at different levels of abstraction to solve the task in a single attempt:

1. The *df* DataFrame must be used amongst the other defined variables in the notebook.
2. Lost means the rocket did not land which requires a filter operation on the *did_land_success* column set to *False*.
3. The assumption that landing or not is the only cause of payload loss amongst the other available data columns.
4. The *Date* column needs to be parsed from a string to extract only the year.
5. The *Date* column is used to group the data by year, and the *PayloadMass* column is summed to get the total payload lost each year.
6. The correct syntax and semantics of the Pandas API calls to achieve the desired output.

```
# How much payload has SpaceX lost each year?
df_t = df.loc[:,['did_land_success', 'Date', 'PayloadMass']]
df_t['did_land_success'] = (df_t.did_land_success == False)
df_t['Date'] = df_t.Date.apply(dateutil.parser.parse).apply(lambda x: x.year, 1)
dfty = df_t[df_t.did_land_success][['PayloadMass', 'Date']].groupby('Date')
dfty.sum()
```

Listing 2.1: Sample task from a *New Tasks* notebook based on SpaceX Falcon launch data

Figure 2.6 shows the number of Pandas API calls that were extracted from the ASTs of the reference solutions. We see that the New Tasks have more Pandas API calls on average, with an average of 5 compared to 3 for Existing Tasks. This highlights the fact that New Tasks were designed to be more complex in order to evaluate on more challenging tasks.

Figure 2.5: Tasks per Notebook

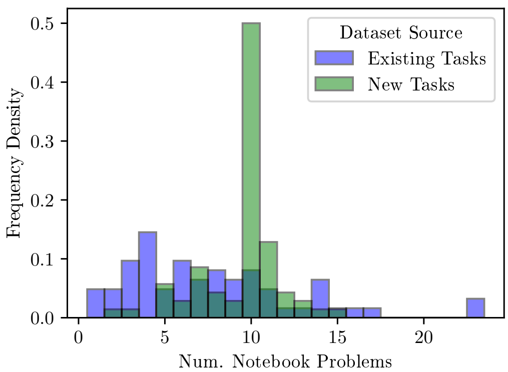
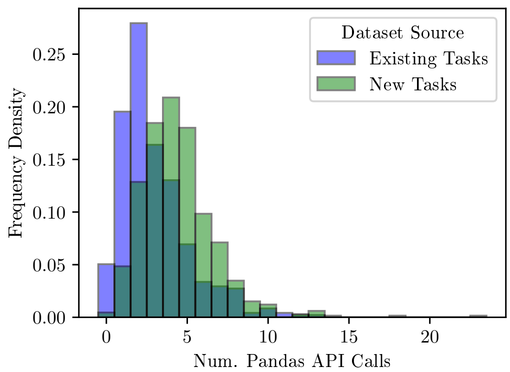


Figure 2.6: Pandas API Calls per Task



The number of iterative tasks per notebook has a median around 10 for both task splits, as shown in Figure 2.5. New Tasks has an unusually high number of notebooks with this many tasks. This perhaps mirrors the fact that these notebooks were created from scratch for the purpose of benchmarking. Regarding evaluation, this does not pose a problem as it is done on a per-task basis and having more notebooks with a high number of tasks asks for a stronger understanding due to the challenge of retaining context and drawing on relevant information from a longer, richer notebook context.

Although task index in a notebook somewhat correlates with the size of the notebook context, a better metric is the number of tokens in the notebook context, shown in Figure 2.7. Tokenization breaks a long sequence into discrete chunks of subword units which is what LLMs process and output. Figure 2.7 uses the Llama 3 tokenizer [38] to tokenize the notebook context for each task. This is a more accurate measure of the notebook context size that the model has to work with. The majority of tasks have a context size below 1500 tokens, with some existing tasks having a size up to 3000 tokens and more. It is important to note that information placement within the context window can impact the effectiveness of retrieval and understanding for in-context learning and this varies based on the model and the specific task at hand [39].

Figure 2.8 shoes the number of DataFrames available for each task. These have been extracted by parsing the notebook context AST and vary from reading data from source files, intermediate DataFrames created during transformations and new DataFrames created for output. New tasks have more variables as DataFrames or series in play on average, with a median of around 15

Figure 2.7: Notebook Context Tokens per Task

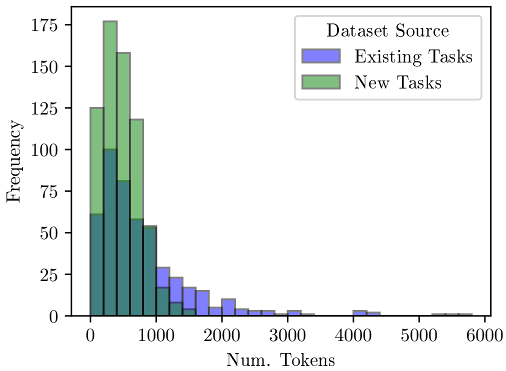
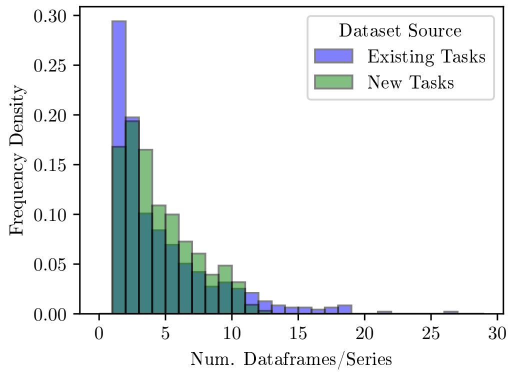


Figure 2.8: Notebook Context DataFrames per Task



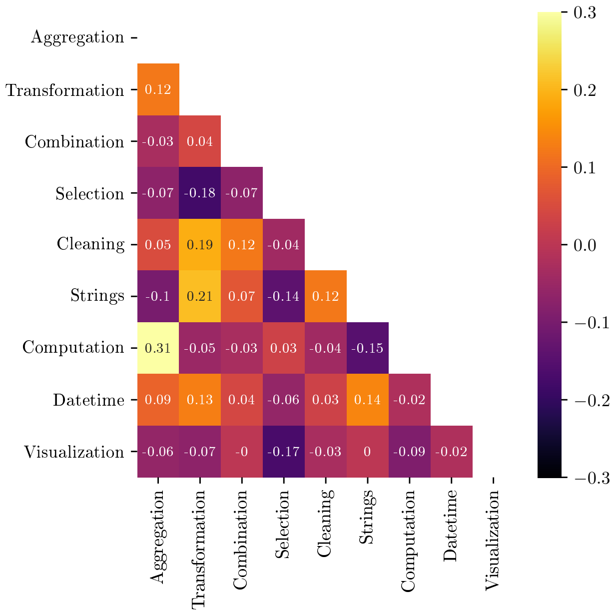
compared to 7 for Existing Tasks. This matches with having larger context sizes which suggests more intermediate DataFrames resulting from more complex transformations and computations.

By grouping Pandas methods into subgroups as shown in Table 2.4, we can then parse the reference solution ASTs to extract methods and determine the requirements of the task. For example, Table 2.4 shows that 42.7% of tasks require aggregation methods like *groupby* and *agg* whilst only 2.7% deal with datetime methods like *to_datetime* and *strftime*. This can be used to understand the distribution of task functionalities that will be important in analysing experiment performance at a more granular level. See Table A.1 in the appendix for a full breakdown of methods. Relationships can be drawn between the method groups to understand how different Pandas method groups occur together and correlate with each other. For example, Figure 2.9 shows that computation methods correlate highly with aggregation, likely due to *groupby* or *agg* being combined with aggregation methods. String operations correlate highly with transformations, perhaps due the need to clean or parse string data before further transformations can be done.

Table 2.4: Pandas Method Groups

Task Type	%	**Methods**
Aggregation	42.7	groupby, agg ...
Transformation	27.8	apply, pivot_table, explode, cut, pct_change ...
Combination	3.0	concat, merge, append, join ...
Selection	68.5	loc, query, nlargest, sort_values, filter, isnull ...
Cleaning	22.4	fillna, rename, drop_duplicates, to_numeric ...
Strings	23.7	extract, startswith, str, replace, contains...
Computation	81.3	max, quantile, value_counts, div, corr ...
Datetime	2.7	to_datetime, strftime, period_range, to_timedelta ...
Visualization	5.9	plot, boxplot, barplot, scatter, hist ...

Figure 2.9: Correlation between Pandas Method Groups in Tasks



2.7 Evaluation Metrics

Evaluation metrics exist in natural language tasks to approximate human judgement through various heuristics as human evaluation of every output is not feasible. Generative models can be evaluated by strict or fuzzy matching of the generated outputs with the reference outputs. Metrics from the machine translation domain have been adapted for code generation tasks to benchmark generative models.

BLEU is a common metric which measures n-gram overlap between the generated and reference outputs to calculate a precision score and incorporates a penalty for outputs that are too short [40]. ROUGE-L is based on recall and that finds the longest common subsequence of n-grams between the generated and reference outputs [41]. METEOR is a metric that uses a harmonic mean of precision and recall of unigrams combined with a penalty for the number of chunks, where a chunk is defined as adjacent mappings between the generated and reference outputs [42]. ChrF calculates an F1-score by averaging precision and recall across character sequences ranging from 1 - 6 in length [43].

However, such similarity metrics do not account for the large space of functionally correct programs that match with the reference solution. CodeBLEU utilises a weighted average of scores derived from data-flow, AST and BLEU to better capture and compare the semantics specific to code where BLEU alone falls short [44]. Programming languages follow strict syntax and semantics and whilst readability is important, functional correctness is paramount and is used in industry in the form of test-driven development. The *pass@k* metric in Equation 2.1 is defined as the by the probability that at least one of the top k samples is correct, given c correct samples out of n total samples [18].

$$\text{pass@k} := \mathbb{E}\left[1 - \left(\frac{C(n-c,k)}{C(n,k)}\right)\right] \tag{2.1}$$

Figure 2.10: The pass@k metric

Correctness can be defined as passing unit tests, strictly matching execution output of the reference solution or fuzzy matching where the output can be semantically equivalent but in different formats or styles such as tables.

Chapter 3

Prompt Design

3.1 System Overview

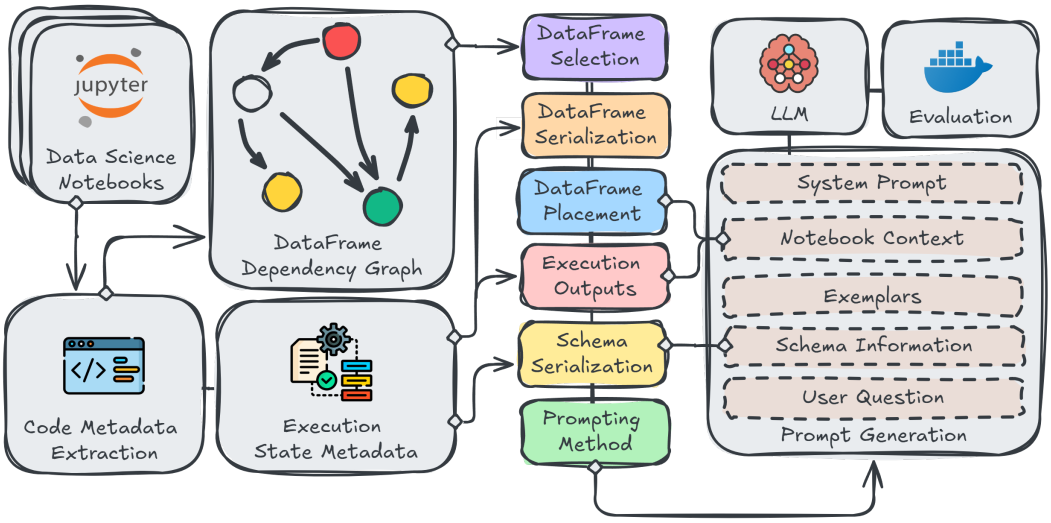


Figure 3.1: Prompt generation pipeline overview

Figure 3.1 shows the end to end pipeline that is used to generate and evaluate prompts for the ARCADE dataset in this work. ARCADE contains notebooks with multiple task cells in each, with each task consisting of the preceding cells as notebook context, followed by a markdown cell containing the user intent and a code cell containing the reference solution.

Each notebook is executed sequentially to collect execution data that is used to serialize DataFrames content and schemas to be included in prompts. For each task in each notebook, we also aggregate preceding code cells that form the notebook context and parse the AST to extract metadata, including DataFrame variables and their lineage. This is used to build a DataFrame dependency graph in order to select relevant DataFrames for inclusion.

We explore various experiment configurations and prompting methods. Starting with simple notebook cells, we build up prompt context with embedded DataFrame outputs, cell execution outputs, serialized schemas and exemplars. We also experiment with prompting techniques such as Chain of Thought (CoT) and Multi-Step prompting. In this section, we dive into each of these components in more detail.

3.2 Execution Metadata

3.2.1 The Importance of Execution Metadata

A core part of working with notebooks is the current execution state. This includes the current Python namespace with imported modules, defined variables, classes and functions as well as outputs from previously executed cells. Although this can be determined by following the previously executed code cells, there is much less room for error and ambiguity if the current state is explicitly provided. The ARCADE dataset assumes all code cells are sequentially executed, and thus the execution state is the same as the state after executing all the code cells before the current one in order.

Pandas variables such as DataFrames and Series go through many transformations and updates in a standard data analysis setting. These operations range from cleaning, formatting and extraction to schema changing transformations like *groupby*, *pivot* and *merge*. Such a variety operations can be difficult to infer from code alone, and a common practice is observing whether the DataFrame is in the expected format in an iterative fashion as the code is developed, such as using *.head()* to output a select few rows. Previous variables are commonly overwritten or updated in place which can make it difficult to track the state of the variable without it being explicitly observed. Therefore, drawing on how humans operate, it seems beneficial for the generative model to have access to the current state of the Pandas data variables and cell outputs that the user has determined is useful, such that it can make more informed decisions.

3.2.2 Collecting Execution Metadata

Since the ARCADE dataset is deterministic and remains static across evaluations, the execution metadata can be collected once and reused for all experiments. The *faketime* library is used to fix the datetime as stated in the original ARCADE paper, to ensure code using relative time logic based on the current datetime deterministic.

For each notebook, an InteractiveShell from the IPython API is used to execute code cells in order. Following each cell execution, all DataFrame or Series variables in the current namespace are recorded, as well as any cell outputs. DataFrame / Series variables in the current namespace are restricted to 100 rows and randomly sampled if the number of rows exceeds this limit, whilst keeping the original index order. This ensures the rows captured are varied and avoids any order bias in the underlying data whilst keeping metadata sizes manageable. Storing a large subset allows for flexibility in formatting later on during prompt design. The same is done for DataFrame / Series variables in cell outputs. All other cell outputs are stored as strings. The set difference from the previous cell execution namespace enables the recording of newly defined variables. Original DataFrame / Series sizes are also recorded to keep as much contextual information as possible.

An important aspect of any such work is to consider the realistic constraints in an environment where such a system would exist in, for example, an auto-complete addon. This means avoiding large latency / resource usage overhead that would make such systems impractical. As part of a LLM data science assistant plugin, direct access to the current execution state namespace and cell outputs is possible via the IPython API which avoids unnecessary overhead from additional cell execution.

3.3 DataFrame Serialization

LLMs take in a string based prompt, which means formatting of variables is important to consider since the goal is to design a prompt that is simple and easy to understand whilst providing as much contextual information as possible. Avoiding unnecessary information is also key to avoid confusion and save on tokens. The space of possible formatting options is vast, therefore intuition based on human readability is used to select formats and constraints. Although certain variations will be discussed, it is not feasible to explore all possible combinations of formatting options. Below are some key factors taken into consideration for serializing DataFrames:

- **Tabular Format** There are many ways of serializing DataFrames, from JSON to CSV to XML. However, in terms of human readability, the default Pandas DataFrame string format is the most intuitive. This has aligned columns and rows, with a header row and index column.
- **Rounding Floats** Rounding all float values to decimals saves tokens by removing unnecessary precision, without losing any context.
- **Truncating Strings** Columns with long strings are truncated to a maximum of 50 chars by preserving the start and end of the string and replacing the middle with an ellipsis to indicate truncation. Columns which contain Python container types like lists, dictionaries or sets are traversed and truncated similarly but to 20 chars to avoid excessive values whilst still providing essential context such as the structure, type and partial semantic information about the data in these columns.
- **Column Types** Column types should ideally be specified to avoid any ambiguity in inferring from column data, especially in string format. This is done by appending the column type to the column name in brackets. Pandas data types include *object*, *int64*, *float64*, *bool*, *datetime64*, *timedelta64* and *category*. The *object* data type covers all other data types which can result in ambiguity as it can be any Python type so in this case, the type is extracted from the first non-null value in the column to provide more granular type information.
- **Limit Rows** The number of rows provided as context can be limited by a constant or token usage, whichever is hit first. It is important to fix the number of tokens used, to provide an upper bound on the total DataFrame tokens used in a prompt.

  The number of rows was fixed to 3 if the original DataFrame size is more than 10 otherwise we try to include all rows. The intuition behind this is small DataFrames of less than 10 rows may contain important relevant information in all rows whilst larger DataFrames may just be raw data with further required processing later on.

  The DataFrame execution metadata is again randomly sampled with the original index order preserved to give a diverse sample of rows to avoid any underlying bias in the data. A constant seed is used for the random sampling to fix the DataFrame string format across dataset generations.

  Each row is iterated through after rounding and string truncation and token usage is calculated and limited. If even a single row is unable to be shown, the row string is truncated instead. If this still exceeds the maximum tokens, only the column information remains. This level of granularity ensures that as much context as possible is included whilst keeping the token usage in check.
- **DataFrame Sizes** If the original DataFrame size is more than what is possible to show, the message "[showing N sample rows out of M rows]" is prepended to the string, where N is the number of rows shown and M is the original DataFrame size. This ensures the LLM is aware of the original DataFrame size. If even a single row is unable to be shown, the message "[omitted due to size]" is appended instead.
- **String Quotes** In string format, differentiating between strings and numeric values is not possible. Whilst column types are provided, to avoid any confusion, all string values are enclosed in single quotes. This also helps in differentiating between strings and other data types like lists, dictionaries and string versions of the same, which may in fact have to be parsed.
- **DataFrame Placement** It is crucial to include the latest state of notebook variables. Therefore, there are two options. The simplest is to place all currently defined variables that are available for the LLM to utilise, at the end of the notebook context.

  The other option is to place the variable information amongst the code cells disguised as cell outputs. It is interesting to see many cases where after a series of operations, the user may output the DataFrame to observe the changes made with the use of *df.head()* or *print(df)*. Parsing ASTs of code cells, it is possible to identify the cell where the last modification to a DataFrame was made and append a code cell with *"print(df)"* and corresponding cell output as the formatted DataFrame string. This embeds variable information in the notebook context in a more natural way, without breaking the flow of the prompt, which is much more intuitive.

These considerations greatly improves on the readability and contextual information present in the serialized format, and better reflects the state of the Pandas variables in a notebook environment.

Listing 3.1 shows the new DataFrame serialization format alongside the original DataFrame schema format for comparison.

```
# In[]:
print(df_anim)

# Out[]:
[showing 3 sample rows out of 1812 rows]
                              theme (list)            name (str)  rating (float64)
124      ["Historical", ..., "Gag Humor"]             "Gintama"              9.05
1303                            ["School"]      "Kareshi  Jijou"              7.60
1488                      ["Mahou Shoujo"]  "Delicious Precure"              7.24
```

Listing 3.1: New DataFrame Serialization Format

```
# Columns in df_anim with example values:
# theme ([School]), name (Kareshi  Jijou), rating (7.60)
```

Listing 3.2: Original DataFrame Schema Format

3.4 Selecting DataFrames

It is crucial to limit the number of DataFrames included in the prompt to provide an upper bound to the number of tokens used for variable information. Not all DataFrames are relevant as some may be intermediate steps or temporary variables. Therefore, a set of criteria is required to select the most relevant DataFrames to include in the prompt. The simple option would be to select those recently defined, as intuitively, notebooks being worked on sequentially would have the most relevant variables just above the current cell. However, this is not guaranteed and might not include required variables defined much earlier in the notebook. This would especially be a problem for tasks further down the notebook.

A more sophisticated approach is to parse the preceding notebook context of a task to extract all defined DataFrames. A dependency graph can be constructed which records data variables, parent variables that it was derived from, and the method used to create the variable. This allows for a better insight into the creation of DataFrames without having access to any semantic relevance and understanding of the DataFrames for the task.

Any DataFrames loaded from external sources should ideally be included as all other DataFrames can theoretically be derived from these. This means those loaded with *read_csv* or *read_json* or other similar Pandas methods, prefixed with *read*. Other DataFrames that should ideally be included are those with complex schema changes and is derived from multiple DataFrames such as with transformation / aggregation functions like *groupby*, *pivot* or *join*.

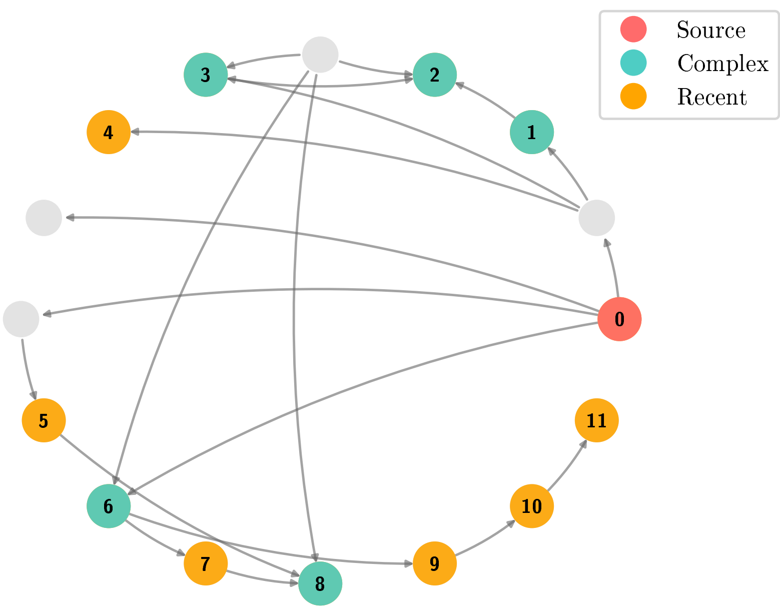
Leaf nodes in the graph suggest DataFrames which end tangential sub-tasks, and therefore could likely be relevant to the task at hand. Also, it is of course impoertant to include some of the most recently defined DataFrames to provide the latest context.

Listing 3.3 shows the notebook code context of a sample task. Building the DataFrame dependency graph for this code in Figure 3.2 provides insights into structure of the notebook and DataFrames within.

```python
from pandas import Series, DataFrame
import pandas as pd

df = pd.read_csv('NYC_Restaurants.csv', dtype=str)

df_noduplicates = df.drop_duplicates(subset='RESTAURANT')
df_notchains = df_noduplicates.groupby("DBA").filter(lambda x: len(x) == 1)
boro_notchain_pivot = pd.pivot_table(df_notchains, index = 'BORO', values = 'RESTAURANT', aggfunc = lambda x: len(x.unique()))
boro_restaurant_pivot = pd.pivot_table(df_noduplicates, index = 'BORO', values = 'RESTAURANT', aggfunc = lambda x: len(x.unique()))
boro_notchain_pivot['TOTAL RESTAURANTS'] = boro_restaurant_pivot

cuis = df_noduplicates['CUISINE DESCRIPTION'].value_counts()
mask = (df['VIOLATION CODE']).isnull()
no_violations = df[mask]
no_violations[['CUISINE DESCRIPTION', 'RESTAURANT']]

cuisine_no_violations = no_violations['CUISINE DESCRIPTION'].value_counts()
mask = (df['VIOLATION CODE']).notnull()
violations = df[mask]
cuisine_violations = violations['CUISINE DESCRIPTION'].value_counts()

total_cuisine = pd.concat([cuisine_violations , cuisine_no_violations], axis = 1)
violations = pd.crosstab(df['BORO'], df['VIOLATION DESCRIPTION']).query('BORO != ["Missing"]')

vstack = violations.stack()
violations2 = vstack.unstack('BORO')
mostcommon = DataFrame({'Most Common Complaint':violations2.idxmax(),'Number of Complaints':violations2.max()})
```

Listing 3.3: Sample notebook context

Figure 3.2: DataFrame dependency graph for Listing 3.3



3.5 AST Parsing

For building the dependency graph from code, the AST of each code line statement is parsed to extract assignments, method calls and method parameters. New DataFrames are determined by the last method of the call chain returning a type of *DataFrame*, *Series*, *NDFrame* or *NDFrameT*. Additional method attributes *iloc*, *loc*, *ix*, *at*, and *iat* were added to this list since they reference slices. Initially, these methods were identified by parsing the return type annotations followed by docstrings of all callable methods in the Pandas library however, this was not consistent or reliable and not all the desired methods were captured. The official Pandas documentation proved to be a better source of information with the desired methods extracted through simple web scraping Listing 3.4. Parent DataFrames are identified from previously defined DataFrames and come from either the base object such as *df* in *df.groupby()* or from the method parameters such *df1* and *df2* in *pd.concat([df1, df2])*.

```python
for module in [pd, pd.DataFrame, pd.Series]:
    methods = get_callable_methods(module)
    for m in methods:
        base_name = module.__name__
        if module.__name__ != "pandas":
            base_name = f"pandas.{base_name}"
        url = f"{base_url}{base_name}.{m}.html"
        response = requests.get(url)
        soup = BeautifulSoup(response.text, 'html.parser')
        elem = soup.find_all(string='Returns')[0]
        return_types = elem.parent.find_next_sibling("dd").dt.text
        if "Series" in return_types or "DataFrame" in return_types:
            res.append({"base": base_name, "method": m })
```

Listing 3.4: Pandas Documentation Scraper

Additional fine-grained metrics can be extracted from any block of code, given code context, such as modified / used DataFrames, any imports and alias imports, user-defined functions and method call chains that can be grouped by modules. These set of AST parsers provide granular metadata about code and can be applied to reference solutions and code cells alike for a more in depth analysis. Listing 3.5 shows examples of the types of metadata that has been extracted from code.

```python
{
    'imports': {'matplotlib.pyplot', 'numpy', 'pandas'},
    'aliases': {'pd': 'pandas', 'np': 'numpy', 'plt': 'matplotlib.pyplot'},
    'dataframes': ['df', 'monthly_metrics', 'result'],
    'used_dataframes': ['df'],
    'modified_dataframes': ['monthly_metrics', 'result'],
    'dataframe_graph': {...},
    'function_calls': [
        'pd.Series',
        'group.sum',
        'group.mean',
        'group.nunique',
        'df.set_index.groupby.apply.reset_index',
        'pd.Grouper',
        'monthly_metrics.strftime',
        'monthly_metrics.sort_values.reset_index'
    ],
    'new_functions': ['calculate_category_metrics'],
    'modules': {
        'pandas': [
            'Series',
            'set_index',
            'groupby',
            'apply',
            'reset_index',
            'Grouper',
            'strftime',
            'sort_values',
            'reset_index'
        ],
    }
}
```

Listing 3.5: Extracted Code Metadata

3.6 Schema Serialization

An alternative or additional method to DataFrame serialization is schema serialization. This omits row data from DataFrames and instead combines schema information into string representation. Type information is extracted similar to DataFrame serialization, with column types appended to the column name in brackets. Many operations work on rows and columns and create new intermediate DataFrames whilst keeping the schema type the same. Therefore, variables can be grouped by schema type and formatted together, as shown in Listing 3.6.

```
DATAFRAMES: orders, cleaned_orders
COLUMNS: Country (str), Customer Name (str), Delivery Year  (float64), Engine (str),
     Model Series (str), Order Month (str), Order Year (float64), Region (str),
    Delivery Total (float64), Order Total (float64), Unfilled Orders (float)

DATAFRAMES: df
COLUMNS: Order Total (float64)

DATAFRAMES: region
COLUMNS: region (str), year (float64)
```

Listing 3.6: Example Schema Serialization

3.7 Prompt Generation

A string prompt is generated for every task in every notebook that forms the ARCADE dataset. Each prompt is constructed based on experiment configurations and can include DataFrames and cell outputs, schema information, additional exemplars etc. in addition to the original serialized notebook context.

For initial experiments, we utilised a simple system prompt, as shown in Listing 3.7, with clear instructions to guide the model to generate desired responses based on the provided notebook context. This is then updated based on prompting technique with specific instructions for Chain of Thought or Multi-Step prompting. These are covered in the following sections.

```
You are a genius Python data science assistant.
Your task is to continue the notebook by answering the user question with the
    provided notebook context.
You must output Pandas Python code that will be parsed and executed in a stateful
    Jupyter notebook environment.
Think carefully about the DataFrames, columns, methods and use sound logical
    reasoning in your response.
You must output your answer in the requested format.
If no format is specified, your output should be a DataFrame or Series.
I will tip $10000000 if your code is clean and correct.
```

Listing 3.7: Default System Prompt

3.8 Chain of Thought

Chain-of-Thought (CoT) is a prompting technique that pushes the LLM to break down the task into sub-problems with intermediates steps and reasoning to reach the final solution. The simplest zero-shot approach is adding "Let's break down this problem and think step by step." to the prompt [23]. This can be extended to a few-shot approach by providing carefully crafted exemplars steps to guide the LLM in the right direction. The original ARCADE paper prepends simple Pandas problems, combined with a step-by-step code breakdown and explanations as exemplars, such as in Listing 3.8. However, these problems are very basic and act as more of a style guide for the output rather than showcasing complex reasoning that some of the more advanced tasks require. See Listing A.1 for a complete sample CoT prompt with execution metadata.

```
# In[]:
# Notebook Context
df_example = pd.DataFrame([
    {'name': 'Peter', 'gender': 'm', 'DOB': '1992/01/17'},
    {'name': 'Amy', 'gender': 'f', 'DOB': '1993/08/30'},
])

# In[]:
# Task:
# How many male and female employees are born in 1992?

# In[]:
# Step 1: convert date of birth in to datetime
df_example['DOB'] = pd.to_datetime(df_example['DOB'])
# Step 2: get the number of male born in 1992
num_male_students = len(df_example[(df_example['DOB'].dt.year == 1992) & (df_example
    ['gender'] == 'm')])
# Step 3: get the number of female born in that year
num_female_students = len(df_example[(df_example['DOB'].dt.year == 1992) & (
    df_example['gender'] == 'f')])
```

Listing 3.8: Sample ARCADE Exemplar for CoT

3.8.1 Bootstrapping CoT

Since we are working with code, by parsing the AST, it is possible to extract details from the provided reference solutions and automatically construct natural language steps. Mentioned DataFrames and corresponding columns can be extracted from the reference code without any complex AST parsing since we already have execution states for each cell. This means we can compare and restrict to only existing DataFrames and only those columns that are present within these DataFrames. There may be certain edge cases, such as when two DataFrames mentioned have same column names, but this is generally a rare case. Additionally, we can extract Pandas methods used for the solution.

With this information, we can construct natural language statements that reference these details as precursor step to generating the code. This can be applied to all previous code cells in the notebook context and gives a greater number of examples then simply prepending fixed exemplars. Although there is no reasoning or explanation provided, this is a crude way to bootstrap CoT steps without spending time on manually crafting or generating exemplars, as this can be a time-consuming process.

This borrows from the idea of automatically generating CoT steps using diverse sampling methods [25]. They also build a bank of exemplars to be inserted into the prompt. In the context of notebooks, there is already a turn-based approach with markdown cells acting as prompts and code cells as responses. This can be leveraged to generate CoT steps with code metadata in the previous code cells, as shown in Listing 3.9.

```
# Step 1:
# I should use 'air'.
# I should use the 'condenser_coil', 'price' and 'brand_name' columns from 'air'.
# I should use the 'unstack', 'dropna', 'groupby' and 'mean' methods.

# Step 2:
air.groupby(['brand_name','condenser_coil']).price.mean().unstack().dropna(subset=['
    Alloy','Copper'])[['Alloy','Copper']]
```

Listing 3.9: Generated CoT Steps

3.8.2 Better Prompting

So far, the system prompt has been quite straightforward. But it is possible to go into much greater detail of what the requirements are for the intermediate steps and invoke the LLM to generate all of its reasoning first before simply translating the instructions into Python code.

Much of the prompt in Listing 3.10 was iterated on using 2 - 3 tasks, observing the output and tweaking it based on what works. Listing 3.11 showcases an example CoT response generate by the Llama 3 70B model.

```
You are a genius AI programming data science expert.
Your goal is to output clear detailed steps such that a 5-year old can follow the steps to write Python code.
The current notebook code and markdown cells are provided in between the <notebook_context> tags.
Notebook variables with column information is provided in the <notebook_variables> tags.

<notebook_context>
...
</notebook_context>

<notebook_variables>
...
</notebook_variables>

My question is as follows:
<question>
...
</question>

For each natural language step, think carefully about the following:
1. All dataframes, series, column names, functions and variables should be in quotation marks.
2. Think carefully about column data types. Observe dataframe contents and data formats. Consider if parsing is required, such as dates. Ensure null/empty values are handled correctly.
3. Think carefully about every column in the dataframe and its relationship with the user's question.
4. Do not make assumptions. You have been given all context required. Ensure your steps match the question requirements.

Write your steps in between <scratchpad> tags.
Write your code in between <python> tags.
Do not output any other text.

Take a deep breath and work on this problem step-by-step.
```

Listing 3.10: CoT Prompt Template

```
<scratchpad>
To find the average division population for different countries having events involving heavy rain, we need to follow these steps:
1. Extract the events involving heavy rain from the 'glc' dataframe.
2. Filter the events to only include those with a 'landslide_trigger' of 'Heavy Rain'.
3. Extract the 'country_name' and 'admin_division_population' columns from the filtered events.
4. Group the data by 'country_name' and calculate the average 'admin_division_population' for each country.
</scratchpad>
<python>
heavy_rain_events = glc[glc.landslide_trigger == 'Heavy Rain']
avg_division_population = heavy_rain_events.groupby('country_name') \
    ['admin_division_population'].mean()
print(avg_division_population)
</python>
```

Listing 3.11: Example CoT Output from Llama 3 70B

3.8.3 Crafting Exemplars

Few-shot prompting with CoT using well-crafted exemplars as demonstrations of reasoning improves performance in various domains [23]. For this task, such exemplars would need to include a question, a *scratchpad* block with natural language steps showing reasoning and the corresponding Python code. As this is a time-consuming process, we only design three exemplars and insert them into the prompt.

Based off of a sample of ARCADE tasks, key requirements for exemplars were identified as follows:

- **Question** An short, ambiguous question that requires breaking down and making certain assumptions to solve.
- **External Imports** Additional libraries such as *dateutil* to highlight such use cases to make outputs simpler and avoid unnecessary code which may be prone to mistakes.
- **Complex Methods** Using a variety of methods to demonstrate a range of operations.
- **Parsing** Include string / date parsing to show how to convert or extract data.
- **Detailed Steps** Detailed step-by-step breakdown of the problem through explaining DataFrames, columns and methods used, along with method parameters and reasoning.
- **Python Code** Python code lines that correspond to the natural language steps.
- **Output Format** Show the expected format to be printing variables without any other extraneous information.

Similar to the system prompt refinement, the exemplars were iterated on and refined based on 2 - 3 outputs. Due to the large space of exemplar designs, it is not possible to evaluate every change. However, we observe small variations in prompt led to large changes in output quality. For example, having numbered steps led to a breakdown in reasoning ability, perhaps due to the model being constrained by the format.

```
<question>
Analyze sales trends across different product categories, identifying categories
    with the highest month-over-month growth and unusual sales patterns.
</question>

<scratchpad>
Import required libraries: 'pandas as pd', 'numpy as np', 'itertools', and 'dateutil
    .parser.parse'.
Parse 'sale_date' column in 'sales_data' using 'sales_data["sale_date"].apply(parse)
    ' to convert string dates to datetime objects.
Create monthly sales by category using 'sales_data.groupby(["category", pd.Grouper(
    key="sale_date", freq="M")])["sales"].sum().unstack(level=0)'.
Calculate month-over-month growth using '.pct_change()' method on the monthly sales
    DataFrame.
Generate category pairs for comparison using 'itertools.combinations(monthly_growth.
    columns, 2)' to create all possible pairs.
Detect unusual patterns by iterating through each category in 'monthly_growth.
    columns' and applying 'detect_unusual_pattern()' function to each category's
    data.
Find top growing categories by calculating average growth for each category using '
    monthly_growth.mean()' and selecting the top 5 categories with highest average
    growth using '.nlargest(5)'.
Print results by printing 'unusual_patterns' and 'top_growing'.
</scratchpad>

<python>
import pandas as pd
import numpy as np
import itertools
from dateutil.parser import parse

sales_data['sale_date'] = sales_data['sale_date'].apply(parse)

monthly_sales = sales_data.groupby(['category', pd.Grouper(key='sale_date', freq='M
    ')])['sales'].sum().unstack(level=0)

monthly_growth = monthly_sales.pct_change()
category_pairs = list(itertools.combinations(monthly_growth.columns, 2))
```

```
unusual_patterns = {{cat: detect_unusual_pattern(monthly_growth[cat]) for cat in
    monthly_growth.columns}}
top_growing = monthly_growth.mean().nlargest(5)
</python>
```

Listing 3.12: Custom CoT Exemplar

3.9 Multi-Step Prompting

Self-debugging has been shown to be effective with the model reflecting on its solution to verify correctness and output format [16]. Additionally, execution results such as outputs or error traces can be provided for additional context. This combines the idea of self-debugging with CoT to create a two-step prompt which pushes the model to think about the problem in a structured way, reflect on its solution and verify correctness of its own solution. Similar to the CoT prompt, this instructs the model to output a detailed analysis of its own solution with respect to the question requirements and to explain the underlying reasoning in the <analysis> tags, in place of <scratchpad>. Listing 3.13 and Listing 3.14 shows the prompts and templates involved in our multi-step prompting. Listing 3.15 provides a step-by-step example of the multi-step prompt chain, including initial prompt, model response, follow-up prompt with error feedback and finally, the corrected model response.

```
Output an analysis of your answer in the <analysis> tag.
Explain the underlying reasoning that the question is asking for.
Explain whether this is truly reflected in your answer.
Take a deep breath and work on this problem step-by-step.
If you believe your answer is correct, do not make unnecessary changes.
If you believe your answer is incorrect, explain what needs to be changed to ensure
    correctness.
Output the final code in the <python> tag.
```

Listing 3.13: Reflection Prompt

```
Your solution was executed in the stateful Jupyter notebook environment.
Here are the execution results of the code you submitted:
<results>
...
</results>
...
```

Listing 3.14: Output Feedback Prompt

```
##################
# Initial Prompt #
##################
[...]
Convert Levy and Mileage columns from object to integer.
[...]

##################
# Model Response #
##################

<scratchpad>
Step 1: Identify the columns that need to be converted from object to integer.
In this case, the columns are "Levy" and "Mileage".

Step 2: Remove any non-numeric characters from the "Levy" column.
We can use the "str.replace" method to remove any non-numeric characters.

Step 3: Convert the "Levy" column to integer.
We can use the "pd.to_numeric" function to convert the column to integer.

Step 4: Remove any non-numeric characters from the "Mileage" column.
We can use the "str.replace" method to remove any non-numeric characters.

Step 5: Convert the "Mileage" column to integer.
We can use the "pd.to_numeric" function to convert the column to integer.

Step 6: Assign the converted columns back to the original dataframe.
```

```
</scratchpad>
<python>
df['Levy'] = df['Levy'].str.replace('[^0-9]', '', regex=True).astype(int)
df['Mileage'] = df['Mileage'].str.replace('[^0-9]', '', regex=True).astype(int)
</python>

########################################
# Follow-Up Prompt with Error Feedback #
########################################
Your solution was executed in the stateful Jupyter notebook environment.
An error occurred during execution of the code you submitted:
<error>
RuntimeError: ValueError: invalid literal for int() with base 10: ''
</error>
[...]

############################
# Corrected Model Response #
############################
<analysis>
The error occurred because the code attempted to convert an empty string to an
    integer, which is not a valid operation. This is likely due to the presence of
    missing or empty values in the "Levy" or "Mileage" columns.

The question is asking to convert the "Levy" and "Mileage" columns from object type
    to integer type. However, the original code did not properly handle the
    possibility of missing or non-numeric values in these columns.

Upon re-examining the problem, it is clear that the code needs to handle these cases
     to ensure correctness. One way to do this is to use the "errors" parameter of
    the "pd.to_numeric" function to specify how to handle errors during the
    conversion process.
</analysis>
<python>
df['Levy'] = df['Levy'].str.replace('[^0-9]', '', regex=True).apply(pd.to_numeric,
    errors='coerce')
df['Mileage'] = df['Mileage'].str.replace('[^0-9]', '', regex=True).apply(pd.
    to_numeric, errors='coerce')
</python>
```

Listing 3.15: Multi-Step Prompt with Error Feedback

Chapter 4

Experiments

4.1 Models

The LLM landscape is constantly evolving with new models and model finetunes being released every week. For this project, the key constraints were model size and hosting providers. Therefore, the Llama 3 family of models was chosen for the experiments, with the hosting provider being Deepinfra. Due to the fast moving nature of this field, we had access to much stronger, smaller models with longer context lengths since the time of the original paper. Model comparison is shown in Table 4.1.

The Llama 3 models released by Meta, have been trained on a huge corpus of 15T tokens and have a context length of 8K [38]. Both models show strong performance across various benchmarks, respective to their size. For the experiments, the instruction tuned models were used, as they offer turn based completion.

Model	No. Params	No. Training Tokens	Context Length
PaLM	62B	3.6T	2048
Llama 3 70B	70B	15T	8192
Llama 3 8B	8B	15T	8192

Table 4.1: Model Comparison

Local deployment was not feasible due to cost / resource constraints. Quantization is a method of pruning model weights to reduce the model size and computational requirements. Although it is possible to run a quantized version of the smaller Llama 3 8B parameter model on a single Nvidia T4 GPU, it has been shown quantization can lead to a significant drop in performance [45]. Additionally, due to the added overhead of maintenance and deployment, it was essential to use a hosted solution.

4.2 Setup

Deepinfra has a rate-limit of 200 concurrent requests per model. Therefore, generating predictions for each model in separate parallel processes, with individual thread pools of up to 200 workers maximised throughput for each experiment.

Deepinfra hosts models at full FP16 precision which gives the best possible performance for the given model. Temperature was fixed to 0.6 since it had the highest pass@5 score on the baseline dataset across both models. Max tokens was fixed to 512, just like the original paper. This is more than sufficient for the output reference solutions. Other parameters such as top_k, top_p, were left at default values. These can have significant impact but prompting was the focus of this project rather than LLM parameters so this has been left for future work.

For evaluation, the original suite of tools was used. For every evaluation job, a pre-built Docker image ensured that environments were consistent across all experiments. Jobs were split between 6 containers with 2 worker threads each, which offered a good balance between throughput and resource utilisation for the VM instance used. Fuzzy matching was used to compare the generated and reference solutions outputs, following execution. Majority voting was used to cluster similar solution outputs together.

Fuzzy Matching is used to compare and verify correctness by comparing the generated solution with the reference solution. Partial matching enables complex DataFrame variables to be matched. Intuitively, the solution is deemed correct as long as all columns and corresponding entries in the reference solution are present in the generated solution. Empirically, the original paper found this heuristic had a low false-negative rate with verifying solutions with alternate but correct output structures.

Post-processing of model outputs is required to extract only the code for execution. When the model follows the requested output format, the code block is easily extracted from between the tag delimiters specified in the prompt. However, we found many edge cases which needed to be handled such as alternative delimiters being used or the end tag being omitted. This was more common in the 8B model outputs, which reflects its limited capacity and less robust instruction following. A straightforward regex operations handled most cases, as shown in Listing 4.1. Occasionally, the model would output multiple blocks of code, however this was skipped since it was a large deviation from the expected output format.

````
def extract_code_from_response(text):
    pattern = r'```(?:python\s+)?(.*?)(?:```|$)|<code>(.*?)</|<python>(.*?)(?:</|$)'
    match = re.search(pattern, text, re.IGNORECASE | re.DOTALL)
    if match:
        code_block = next(group for group in match.groups() if group)
        return code_block.strip(), text
    return text, text
````

Listing 4.1: Code Extraction

4.3 Results

Table 4.2: pass@5 evaluation results on the ARCADE dataset

Experiment Name	Existing Tasks		New Tasks	
	Llama 3 70B	Llama 3 8B	Llama 3 70B	Llama 3 8B
Baseline Vanilla + FS	54.2	48.5	34.3	24.2
Baseline CoT + FS	-4.0 50.2	-23.9 24.6	+0.9 35.2	+16.3 7.9
+ Explanations	-1.9 48.3	+8.0 32.6	-5.0 30.2	+8.7 16.6
Vanilla	47.2	44.3	23.3	17.1
+ DataFrames	+4.7 51.9	+7.6 51.9	+8.9 32.2	+7.7 24.8
+ Execution Outputs	+2.5 54.4	+2.3 54.2	+0.5 32.7	+3.6 28.4
+ Exemplars (ARCADE)	+2.3 56.7	-2.5 51.7	+0.4 33.1	-0.3 28.1
+ Schema Information	+0.9 57.6	+6.7 **58.4**	+5.0 38.1	+2.3 30.4
CoT (*)	58.8	50.8	40.1	23.3
+ Schema Information	-0.4 58.4	+3.2 54.0	+2.1 42.1	+4.4 27.7
+ Exemplars (Custom)	-1.0 57.4	-5.7 48.3	-1.3 40.8	-2.7 25.0
CoT + 2 Step (*)	**61.3**	55.3	**44.9**	32.1
+ Errors (**)	-0.4 60.9	+0.0 55.3	-0.6 44.3	+3.4 **35.5**
+ Results	-1.9 59.0	+0.6 55.9	-0.1 44.2	-2.1 33.4

(*) includes DataFrames and Execution Outputs
(**) requires execution feedback

Table 4.2 shows the results of the experiments conducted on the ARCADE dataset, with different prompt formats and configurations. The experiments are divided into Existing Tasks and New Tasks, and the pass@5 metric is reported for the Llama 3 70B and Llama 3 8B models.

Updated architecture, training techniques and tokens results in much stronger models, with the smaller 8B already beating the original PaLM 64B model. Baseline experiments enables us to set a benchmark for successive experiments with Llama 3 models.

Vanilla refers to the completion prompt format where the LLM is prompted to complete the notebook context with the last markdown cell containing the task description. ARCADE exemplars are the same as those used in the baseline experiments denoted by FS (Few Shot). They contain step-by-step breakdown of the task with commented explanations.

CoT refers to the Chain of Thought prompt format where the LLM is prompted to write out detailed steps in between <scratchpad> tags, breaking down the question into manageable steps. Following this, the steps are essentially translated into executable Python code by the model.

The two-step prompt format expands on CoT by asking the model to reflect on its solution and correct if required. It is prompted to reflect and analyse its previous solution in <analysis> tags, before outputting corrected code in <python> tags, similar to before. Error and execution output feedback requires execution of the solutions, in order to generate to follow up prompt for the second iteration.

Figure 4.2 shows the different errors encountered during evaluation. Logical errors refer to solutions evaluated incorrect. Parse errors refer to code unable to be executed due to incorrect syntax, often caused by problems during extraction. Python Runtime errors can be further broken down into different errors. Figure 4.3 shows the frequency of Python Runtime errors across the experiments. Listing 4.2 shows a few examples of runtime errors encountered for a single task.

```python
# Question: Which stock had the highest monthly volatility each month? Show the stock name and volatility.

# RuntimeError: TypeError: argmax() got an unexpected keyword argument 'level'
volatility.idxmax(level=1).reset_index(name='volatility').sort_values('Date')

# RuntimeError: AttributeError: 'tuple' object has no attribute 'reset_index'
volatility.idxmax(axis=0).reset_index(name='volatility').sort_values('Date')

# RuntimeError: IndexError: positional indexers are out-of-bounds
volatility.reset_index().groupby('Date').idxmax().apply(lambda x: volatility.iloc[x]).reset_index(drop=True)[['Security', 'Close']]
```

Listing 4.2: Sample Runtime Error Responses

4.3.1 Baseline Experiments

Llama 8B performs relatively poorly on the baseline CoT and CoT with explanation runs. Due to lack of clearly defined instructions combined with the few-shot exemplars, the model seems to struggle to output a valid response. Figure 4.2 shows that logical and runtime errors both remain quite high which suggests that the model is confused by the additional explanations as comments. Many cases where the post-processing extraction expected a single block of code, the model outputted multiple blocks of code and explanations without adhering to the notebook format. This issue is later mitigated by clearly defined instructions in the system prompt. In general, parse errors have dramatically reduced with later experiments for both models.

Llama 70B does not have this problem and can infer and correctly output responses in the right format for the baseline experiments without any explicit instructions, as seen in the original paper.

~~~~
Let's solve this problem step-by-step.
Step 1: Filter flights_df to include only 2013 data
```
df_2013 = flights_df[flights_df['year'] == 2013]
```

Step 2: Filter df_2013 to include only flights from NYC airports to Seattle
```
df_nyc_seattle = df_2013[(df_2013['origin'] == 'JFK') | (df_2013['origin'] == 'LGA') | (df_2013['origin'] == 'EWR') & (df_2013['dest'] == 'SEA')]
```

Step 3: Count the number of flights
```
num_flights = len(df_nyc_seattle)
```
So, the number of flights from NYC airports to Seattle in 2013 is `num_flights`.
~~~~

Listing 4.3: Llama 8B Incorrect Output Format for Baseline CoT

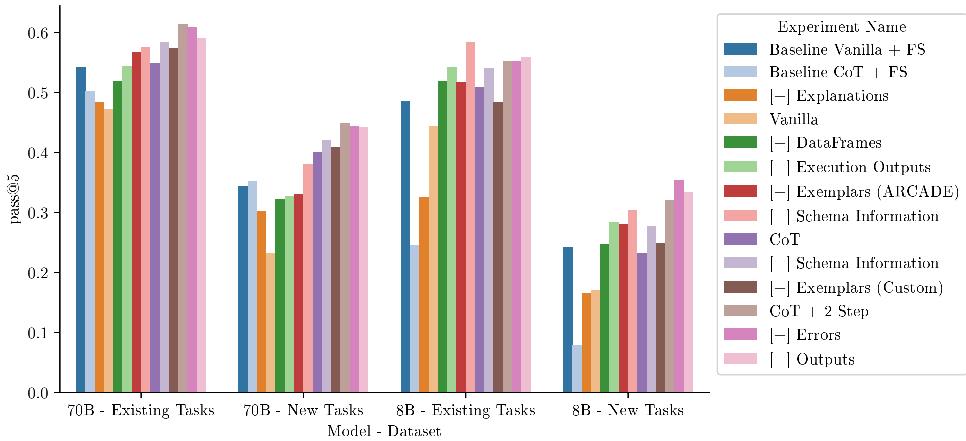


Figure 4.1: pass@5 evaluation results on the ARCADE dataset

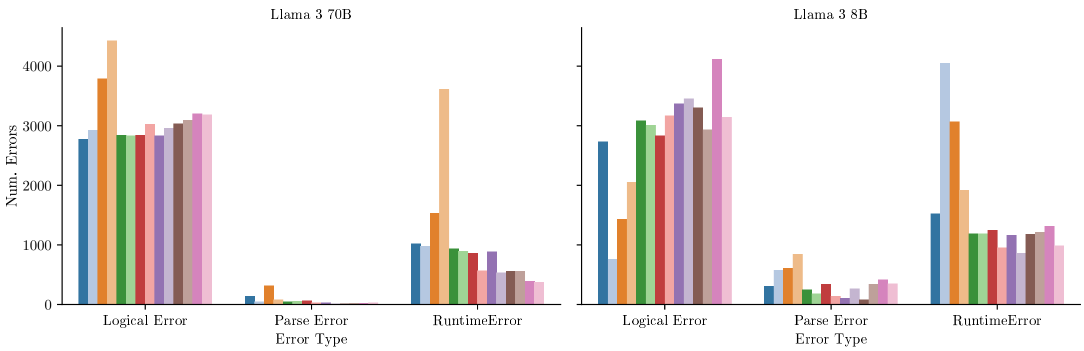


Figure 4.2: Frequency of Evaluation Errors

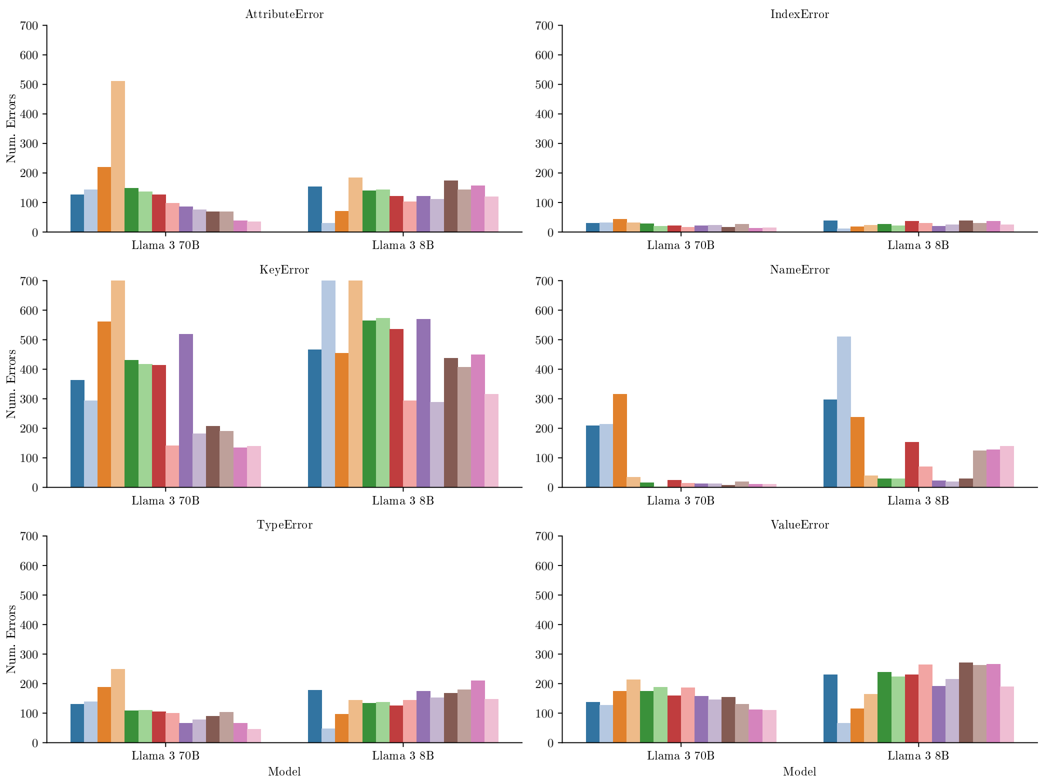


Figure 4.3: Frequency of Python Runtime Errors

4.3.2 Vanilla Experiments

Vanilla on its own only consists of the cleaned notebook context, where consecutive markdown cells are collapsed and empty cells are removed such that the cells alternate between code and markdown. It does not have any schema information unlike the baseline experiments. However, it includes a system prompt with simple clear instructions.

We observe a large jump in errors in Figure 4.2, which is expected due to lack of visibility into DataFrame variable contents. This is reflected in the jump in logical errors, as well as high number of *KeyErrors*, *AttributeErrors* and *TypeErrors* where the incorrect columns are being accessed, as shown in Figure 4.3. However, the 8B model shows a strong performance in comparison to the previous baselines, highlighting the importance of a strong system prompt.

DataFrames, followed by Execution Outputs, shows improvements in performance for both models. For the 8B model, this jump is significant with the Existing Tasks pass@5 matching the 70B model. The 70B model also shows a significant improvement in performance, with the New Tasks pass@5 score increasing by 9.5 points in Figure 4.1.

Exemplars degrade the performance of the 8B model, which can be attributed to the increase in parse errors and runtime errors, as seen in Figure 4.2. Manual exploration shows parse errors are caused by outputs such as uncommented explanations, hypothetical cell outputs and other extraneous text that meant the code section could not be correctly extracted. See Figure A.1 for the Gradio UI built for such exploration. The high rise in *NameErrors* in Figure 4.3 resulted in the incorrect use of *df_example*, which was a variable in the exemplars, but not in the notebook context. Unlike the 70B model, the 8B model seems confused by the exemplars. Like the original baseline experiments, exemplars were simply prepended to the notebook context which may have been the reason for the confusion.

Schema information boosts the Llama 3 8B model significantly relative to the 70B model. Smaller models have difficulty inferring complex relationships therefore, schema information proves simple clarity to variables, column names and types without the need for additional rows. Figure 4.3 shows a large drop in runtime errors, specifically *KeyErrors* for the 8B model.

4.3.3 Chain of Thought Experiments

CoT improves the 70B model, with New Tasks showing a significant jump. CoT pushes the model to explicitly reason about its steps and breakdown the problem into step-by-step instructions, before translating this into code. The preceding *<scratchpad>* provides a clear thread of reasoning that incrementally builds with explicit activation of knowledge. This leads to more accurate solutions with fewer logical errors, as seen in Figure 4.2. New Tasks benefits more from this to higher complexity on tasks that naturally require a multi-step approach.

However, the 8B model drops in performance. Although it follows the format structure, it often produced illogical steps which led to worse performance. Smaller models don't seem to have the capacity to correctly reason and breakdown the problem into steps.

With additional crafted exemplars, both models drop in performance. Although *ParseErrors* are reduced for 8B model, logical and runtime errors increased for both models. This suggests exemplars interfering with reasoning steps and constraining flexibility in the model's outputs With existing notebook context already providing example code, further studies are required into designing exemplars.

Table 4.3 shows results for with generated exemplars. Extracted metadata from code cells was used to generate crude natural language statements as a step before generating the code. This was a method to bootstrap contextually relevant exemplars without manually crafting them. The 70B model shows an increase in performance from the vanilla experiment without exemplars across all tasks. However, the original ARCADE exemplars, followed by the addition of schema information both outperform the generated exemplars. It seems the generated exemplars are too simplistic and do not demonstrate underlying reasoning, however they do extract key information as precursor to generating code. This reduces the number of *NameErrors* but ultimately does not perform as well as the ARCADE exemplars. Manual exploration shows that model outputs often don't even follow the generated exemplars format, which suggests further tweaking of the system prompt is required for better response outputs.

Table 4.3: pass@5 evaluation results for generated CoT exemplars

Experiment Name	Existing Tasks		New Tasks	
	Llama 3 70B	Llama 3 8B	Llama 3 70B	Llama 3 8B
Vanilla (*)	54.4	54.2	32.7	28.4
+ Exemplars (ARCADE)	56.7	51.7	33.1	28.1
+ Schema Information	**57.6**	**58.4**	38.1	**30.4**
Vanilla (*) + Exemplars (Generated)	55.0	49.8	34.8	24.3
+ Schema Information	57.3	50.6	**38.7**	25.6

(*) includes DataFrames and Execution Outputs

4.3.4 Multi-Step Experiments

Simply asking the model to reflect and analyse its own solution while correcting if required gives another large boost in performance across all models and tasks. This iterative refinement allows the models to catch inconsistencies, potential errors and output format issues after which it can rewrite the solution. However, this also results in degradation for correct solutions being updated and evaluated incorrect.

For the smaller 8B model, this is often due to confusion due to limited capacity for effective reasoning. The 70B model often falls into the trap of overthinking and overcomplicating the solution, which then leads to incorrect evaluation, even though the solution may be logically correct with the given context.

Although we can validate whether a solution is correct by comparing execution outputs with the reference solution, this is not realistic and the model does not have access to this type of feedback. Therefore, even correct solutions go through the second step to test the model's robustness in generating correct solutions.

With execution, it is possible to record runtime errors and the execution outputs. By passing in the error thrown, there is a significant reduction in runtime errors for both models. However, logical errors moved higher instead which suggests in the process of correcting the error the model was unable to correct other logical errors in the solution or introduced some instead and perhaps, another refine iteration is required for these solutions. Although one would expect performance to remain the same since the prompts only give additional context to help the model reflect on its solution, outputs seem to distract the models.

Figure 4.4 shows the change in errors between solutions from the original first step and updated second step of the multi-step experiments. [+] Errors and [+] Outputs refer to experiments with the additions of error messages and solution output feedback respectively. Each column of each grid breaks down the percentage of original solutions results that were evaluated as correct, logical, runtime or parse errors in the second step.

Figure 4.4 shows 82% of the 70B model solutions are still correct after the rewrite, with 14% turning into Logical Errors. On the other hand, only 55% of the 8B model solutions are still correct after the rewrite, with 36% turning into Logical Errors. This difference highlights the limited capacity of the smaller model to correctly reason and ends up making more mistakes in the process due to added confusion.

Runtime error feedback corrects 18% of runtime errors compared to 14% without, for the 70B model. For the 8B model, this is 22% compared to 16% without. Explicit error feedback seems to help the smaller 8B model more on New Tasks, likely due to providing crucial context that the model might struggle to infer on its own, given the increased problem complexity.

Llama 8B parse errors are much higher than the 70B model, and the multistep approach seems to mitigate this. However, as mentioned, due to its limited capacity, only 10% is converted to correct, whilst more than 60% convert to logical or runtime errors.

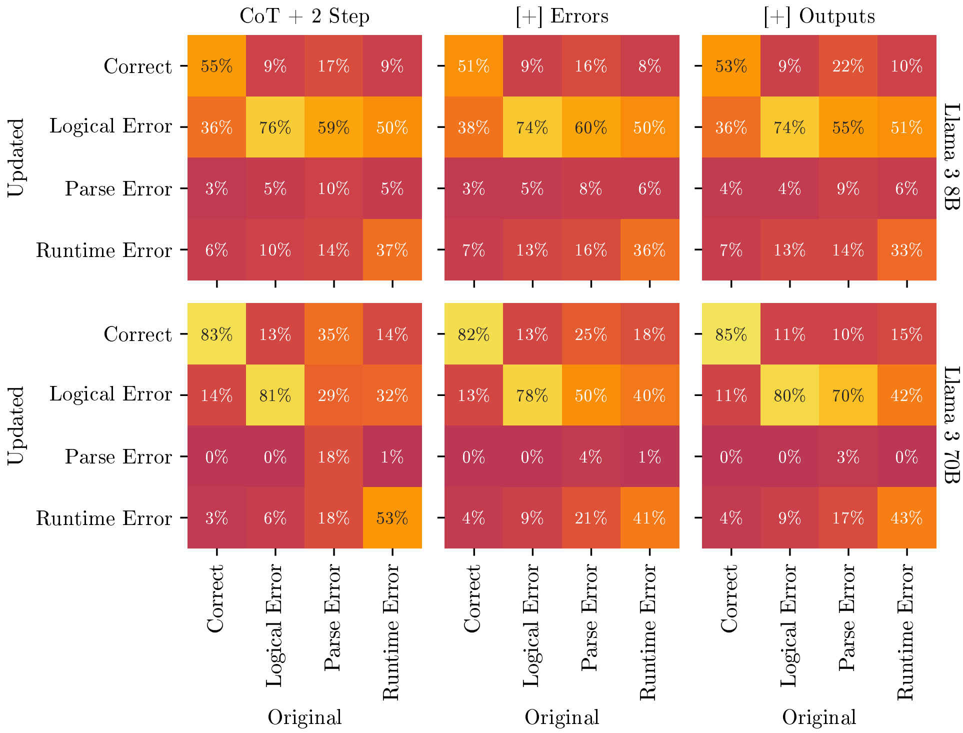


Figure 4.4: Updated solution results for Multi-Step experiments

4.4 Self-Consistency

Self-consistency is achieved through majority voting with the maximum number of solutions in a cluster determines the selected solution. Clustering is based on the execution outputs with identical outputs grouped together. This allows the selection of the most consistent answer out of a range of generations. Figure 4.5 shows the number of cluster for each experiment. Lower number of clusters reflect greater consistency in outputs. We observe most solutions for the 70B model 1 - 2 clusters before a steep decline, whilst the 8B model has a more gradual decline with a flatter distribution of clusters across tasks. This highlights the capacity and consistency of the larger model indicates a higher confidence in the generated solutions. The smaller model has more variables in its outputs reflecting its lower capacity and tendency to make more mistakes resulting in a wildly different execution output.

Across experiments, we see CoT and multi-step resulting in a flatter distribution of clusters. CoT naturally introduces more variations in the reasoning path taken which results in more distinct clusters in the outputs. Multi-step heightens this effect and results in a more diverse range of solutions and clusters. The smaller 8B model inferior capacity for reasoning is reflected in the multi-step experiments and larger number of clusters.

In a realistic environment, self-consistency by majority voting is useful for selecting an answer out of a diverse range of outputs. This is essential where execution is not possible or in domains where correctness cannot be easily determined. We can plot the accuracy of self-consistency decoding by determining whether all solutions in the largest cluster are correct.

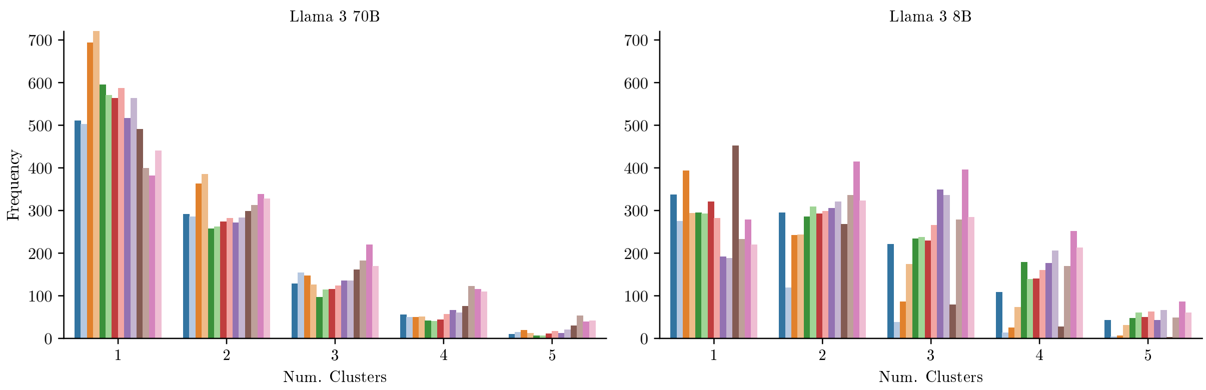


Figure 4.5: Frequency of Response Clusters per Task

4.5 Task Type Breakdown

Pandas methods can be grouped into categories for common functionality, such as aggregation or computation. By parsing the AST of reference solutions, we can determine the type of task being performed and group them accordingly with a priority order if there is overlap. Figure 4.6 shows the pass@5 scores for each task type. Some groups such as cleaning or string operations have been omitted due to low sample size. See Table A.1 for full breakdown of task types.

Scores follow a similar pattern to overall results with the 70B model outperforming the 8B model across all task types and CoT and multi-step incrementally performing better for the 70B model. Aggregation and transformation tasks performs worse than computation and selection tasks for both models.

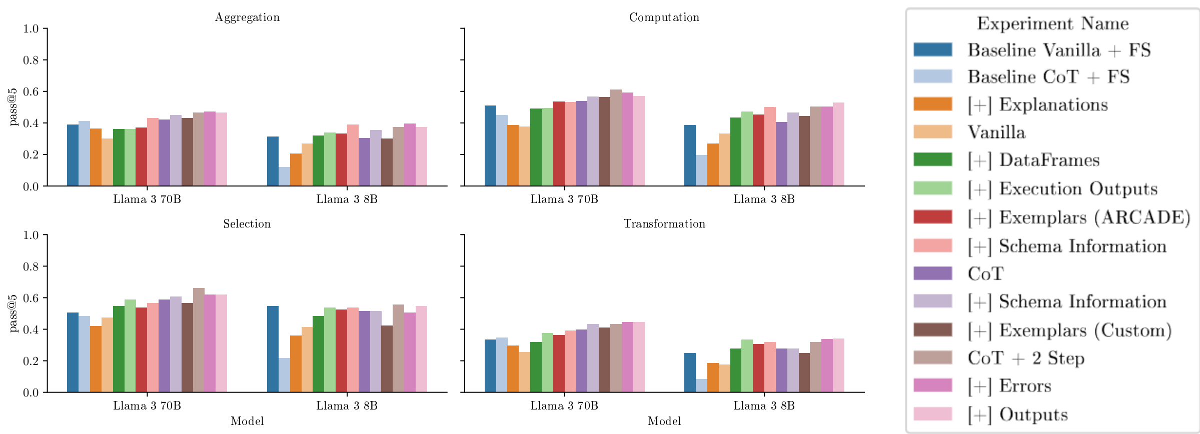


Figure 4.6: pass@5 Task Type results breakdown

4.6 Task Complexity

The number of Pandas API calls in the reference solutions can be used as a proxy for difficulty of the corresponding task. We observe a clear downtrend in the pass@5 score as complexity increases for both models. Within the experiments, the CoT followed by multi-step experiments show increasing performance as complexity increases.

The sample size for tasks with reference solutions having 20-24 Pandas API calls is small however, we see only certain experiments such as CoT and multi-step achieving high scores for the 70B model. This reflects the model's capacity to reason and breakdown complex tasks and also highlights the benefit of self-reflection for models.

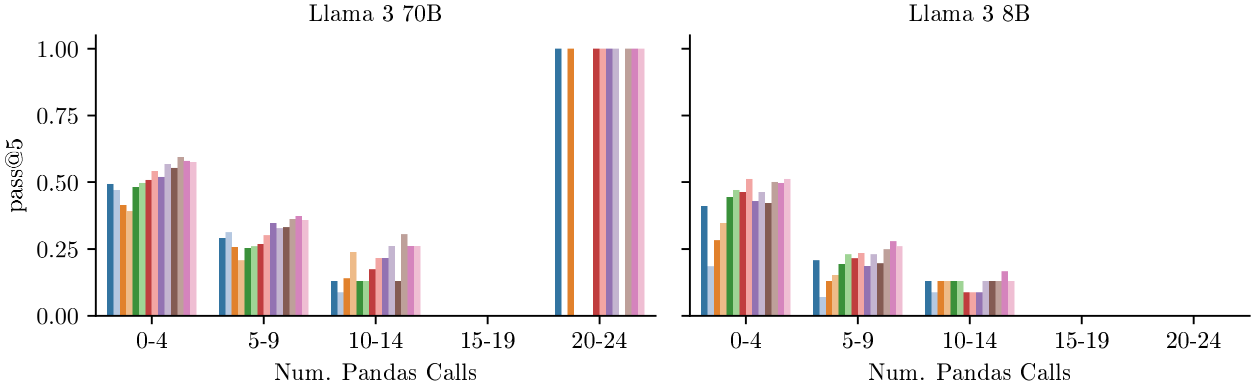


Figure 4.7: Pandas API Calls in Reference Solution

4.7 DataFrame Placement

DataFrame placement refers to the position and format of serialized DataFrame / Series variables in a prompt. Inline DataFrames are placed after their last modification in the notebook code context cells, with a print statement followed by a cell output with the serialized string. This format is used for all experiments. Appended DataFrames are placed after the notebook context, within *<notebook_variables>* tags. Table 4.4 shows the pass@5 scores for the CoT experiment inline and appended DataFrame placement configurations.

We find that DataFrame placement has a significant impact on the 70B model's performance. Inline DataFrames naturally follow the notebook format with inserted code cells and cell outputs. This also reflects human behaviour where DataFrames are often printed after operations to observe the state. Inline placement puts variable data right after relevant operations and better simulates real-life Jupyter notebooks with execution results are embedded within the notebook, highlighting the code-data relationship. Relevant context is grouped together and reduces context switching for the model, leveraging the model's superior context handling and higher capacity. It seems the smaller model is unable to utilize the code-data relationship effectively and therefore, does not benefit as much from inline placement.

Table 4.4: pass@5 evaluation results comparing DataFrame placement

Experiment Name	**Existing Tasks**		**New Tasks**	
	Llama 3 70B	Llama 3 8B	Llama 3 70B	Llama 3 8B
CoT (Inline DataFrames)	**58.8**	**50.8**	**40.1**	23.3
CoT (Appended DataFrames)	56.5	50.6	37.2	**24.8**

4.8 MT Metrics

Since we have a reference solution, we can generate common metrics used in machine translation to evaluate the generated solutions. Figure 4.8 shows the average BLEU, METEOR, ROUGE-L and ChrF scores for the CoT experiment, with individual scores for correct and incorrect tasks as black bars.

We observe some correlation between all metrics and accuracy, with lower scores seen for incorrect tasks across all metrics. However, this difference between Existing Tasks and New Tasks, with Existing Tasks having a much larger gap between scores for correct and incorrect tasks. It may be easier to distinguish between correct and incorrect tasks for the Existing Tasks as these are tasks that are less complex and therefore have simpler solutions that means the difference between right and wrong answers is more larger. It is also likely the models have been trained on Existing Tasks and as a result, are generating outputs similar to the reference solutions, which the models have likely seen before.

We also observe the 8B model has a much larger gap between correct and incorrect tasks compared to the 70B model. This can be attributed to the fact that the 70B outputs are closer to the reference solutions, with subtle correctness issues compared to the 8B model whose outputs have more variability in content and structure.

BLEU has the lowest correlation with accuracy due to not accounting for code structure and semantics, however subtle differences can affect accuracy without the NLP metrics and this shows that such metrics are less than ideal for evaluating in code domains [46].

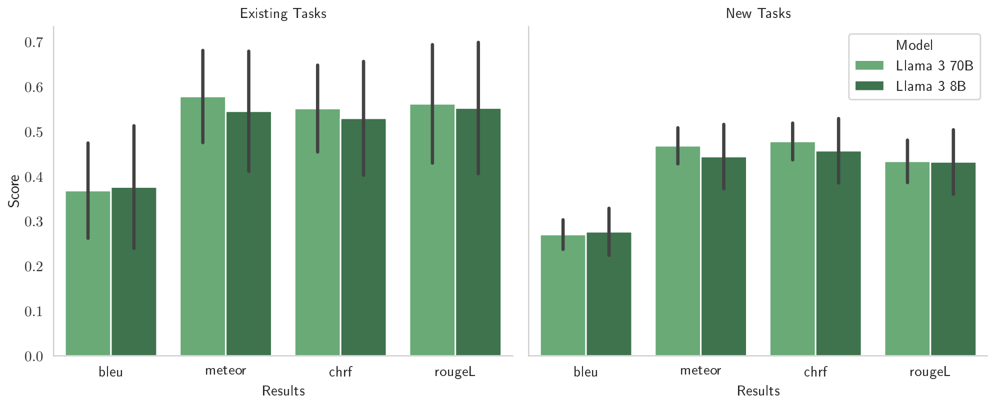


Figure 4.8: MT Metrics for CoT results

4.9 Ablation Studies

We run the CoT experiment with only schema information to understand the impact of execution metadata for prompts. We observe that pass@5 scores drops significantly across both models and task splits, as shown in Table 4.5. This highlights the importance of providing notebook Pandas variables and cell outputs as contextual information when working with computational notebooks. Schema information alone is not sufficient but greatly reinforces variable context when combined with execution metadata.

Table 4.5: pass@5 evaluation results on the ARCADE dataset

Experiment Name	**Existing Tasks**		**New Tasks**	
	Llama 3 70B	Llama 3 8B	Llama 3 70B	Llama 3 8B
CoT (*)	**58.8**	50.8	40.1	23.3
+ Schema Information	58.4	**54.0**	**42.1**	**27.7**
CoT + Schema Information	55.0	49.8	39.3	23.6

(*) includes DataFrames and Execution Outputs

Chapter 5

Conclusion

5.1 Summary

Table 5.1: pass@30 baseline comparisons on the New Tasks dataset

Experiment Name	New Tasks (pass@30)			
	PaLM 62B (†)	PaChiNCo 62B (‡)	Llama 3 70B	Llama 3 8B
Baseline Vanilla + FS	39.8	48.6	-	-
Baseline CoT + FS	-	52.9	-	-
CoT (*)	-	-	52.5	43.3
CoT (*) [Temp. Sampling]	-	-	**54.3**	42.7

(*) includes DataFrames, Execution Outputs and Schemas
(†) fine-tuned on Python
(‡) fine-tuned on Python and Notebooks

5.1.1 Comparison with ARCADE Benchmark Results

Table 5.1 shows pass@30 results to allow for comparison with PaChiNCo results from the original ARCADE paper. PaChiNCo is a 62B parameter model which is derived from PaLM. Fine-tuning was performed in two stages: first on Python code followed by Jupyter notebooks. Similar to the ARCADE dataset, notebooks were sourced from GitHub, deduplicated with Existing Tasks and preprocessed to build a dataset of notebooks. On the other hand, Llama 3 models are trained on a much larger corpus of 15T tokens but without any specialization to a domain.

PaChiNCo achieves a pass@30 score of 52.9 on the Baseline CoT. Llama 3 70B achieves a very close 52.5 score. With varied temperature sampling, this is boosted to 54.3, outperforming PaChiNCo. Llama 3 8B outperforms a fine-tuned PaLM 62B model with a score of 43.3. Unfortunately, the original paper does not provide other results for comparison with other experiments.

Although Llama 3 70B is a stronger model, fine-tuning offers significant boosts in performance. It enables much better understanding of complex relationships within a prompt since it is trained on a domain specific dataset. Modern fine-tune techniques can enable models orders of magnitudes smaller than GPT-4 to score better on domain specialisation [47]. This means it has already seen the notebook format, code structures and natural language / code relationships between the markdown and code cells. This is not true for the Llama 3 models.

5.1.2 Discussion

We show that through well-crafted domain-aware prompting methods utilising best practices and prompting techniques, we can achieve strong performance in the data science notebooks domain without fine-tuning. Fine-tuning is expensive, time-consuming and requires manually building a dataset. It also restricts the ability to leverage the latest model capabilities as fine-tuning is model specific. With only prompting, we can get an instant boost in performance by switching to a stronger model that may have been released.

Significant boosts through the addition of DataFrames, execution metadata and schema information highlights the important of providing as much contextual information as possible to the model, whilst still keeping prompts clean and concise. Additional multi-step prompting offers notable gains in performance highlighting model debugging capabilities through reflection. Although it can fall short in reasoning, Llama 8B scores rebound with additional contextual feedback such as error traces and execution outputs.

Programming assistants are powerful tools for data scientists and developers. We aimed to keep prompt generation feasible and realistic for real-world Jupyter notebook environments. Execution metadata, such as DataFrame variables and cell outputs, can be extracted from the current notebook execution state without additional overhead. While this work assumes sequential execution, it could be extended to restrict context cells or add markers highlighting cell execution order. Simply asking the model to reflect on its solution provides significant improvements.

As performance gains from running models at scale reduce latency overhead and smaller models become stronger, multiple model calls for self-consistency decoding and multi-step prompting become feasible in real-time coding environments. A parallel execution environment could be maintained, sharing certain elements with the user's environment. This would serve as a model scratchpad to identify potential errors and inconsistencies before suggesting solutions.

5.2 Future Work

Further exploration offers numerous avenues. Exemplars at the prompt level remain underexplored. Combined with RAG, we could potentially select task-specific exemplars on the fly, perhaps using a smaller retriever model based on anticipated Pandas methods. A large bank of exemplars, covering diverse tasks, could be crafted or generated by larger models. This approach can guide smaller models in reasoning and task comprehension.

Parameter-Efficient Fine-Tuning (PEFT) methods could be explored to fine-tune adaptors on top of models, enabling the learning of domain-specific prompt structure such as notebook cells and DataFrames. This could potentially offer significant boost in performance for working with notebooks without the need for full fine-tuning.

Multi-step prompting naturally extends to agentic systems, unlocking potential for a more interactive, collaborative environment where the model functions as a partner rather than mere autocomplete. This enables a more powerful system capable of reasoning and reflection, as well as tool use and interacting with the environment. Consequently, we can offload more data science tasks, including data loading from various sources and across different platforms and environments. Off-the-shelf machine learning tools and libraries such as AutoML can be integrated within such a system to offer a more comprehensive data science assistant, balancing semantic understanding and code synthesis capabilities with powerful automated analysis and feature engineering.

Bibliography

[1] Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. Natural Language to Code Translation with Execution. 2022. https://arxiv.org/abs/2204.11454.

[2] Rajas Agashe, Srinivasan Iyer, and Luke Zettlemoyer. JuICe: A Large Scale Distantly Supervised Dataset for Open Domain Context-based Code Generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5435–5445, Hong Kong, China, 2019. Association for Computational Linguistics. https://www.aclweb.org/anthology/D19-1546.

[3] Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. Execution-based Evaluation for Data Science Code Generation Models. http://arxiv.org/abs/2211.09374, November 2022.

[4] Shubham Chandel, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Training and Evaluating a Jupyter Notebook Data Science Assistant. *ArXiv*, January 2022. https://arxiv.org/abs/2201.12901.

[5] Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural Language to Code Generation in Interactive Data Science Notebooks. https://arxiv.org/abs/2212.09248v1, December 2022.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, and Chung et al. PaLM: Scaling Language Modeling with Pathways. http://arxiv.org/abs/2204.02311, October 2022.

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[8] OpenAI Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, and Almeida et al. GPT-4 Technical Report. March 2023. https://arxiv.org/abs/2303.08774.

[9] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent Abilities of Large Language Models. http://arxiv.org/abs/2206.07682, October 2022.

[10] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, and Askell et al. Language Models are Few-Shot Learners. http://arxiv.org/abs/2005.14165, July 2020.

[11] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned Language Models Are Zero-Shot Learners. *ArXiv*, September 2021. https://arxiv.org/abs/2109.01652.

[12] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code. http://arxiv.org/abs/2311.07989, November 2023.

[13] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. http://arxiv.org/abs/2108.07732, August 2021.

[14] Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R. Bowman, Kyunghyun Cho, and Ethan Perez. Improving Code Generation by Training with Natural Language Feedback. 2023. https://arxiv.org/abs/2303.16749.

[15] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large Language Models meet Program Synthesis. http://arxiv.org/abs/2112.02969, December 2021.

[16] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. 2023. https://arxiv.org/abs/2304.05128.

[17] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. http://arxiv.org/abs/2303.17651, May 2023.

[18] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, and Burda et al. Evaluating Large Language Models Trained on Code. http://arxiv.org/abs/2107.03374, July 2021.

[19] Anirudh Khatry, Joyce Cahoon, Jordan Henkel, Shaleen Deep, Venkatesh Emani, Avrilia Floratou, Sumit Gulwani, Vu Le, Mohammad Raza, Sherry Shi, Mukul Singh, and Ashish Tiwari. From Words to Code: Harnessing Data for Program Synthesis from Natural Language. 2023. https://arxiv.org/abs/2305.01598.

[20] Stefan Hegselmann, Alejandro Buendia, Hunter Lang, Monica Agrawal, Xiaoyi Jiang, and David Sontag. TabLLM: Few-shot Classification of Tabular Data with Large Language Models. 2022. https://arxiv.org/abs/2210.10723.

[21] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The Curious Case of Neural Text Degeneration. http://arxiv.org/abs/1904.09751, February 2020.

[22] Gabriel Poesia, Oleksandr Polozov, Vu Le, A. Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. *ArXiv*, January 2022. https://arxiv.org/abs/2201.11227.

[23] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, E. Chi, F. Xia, Quoc Le, and Denny Zhou. Chain of Thought Prompting Elicits Reasoning in Large Language Models. *ArXiv*, January 2022. https://arxiv.org/abs/2201.11903.

[24] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large Language Models are Zero-Shot Reasoners. http://arxiv.org/abs/2205.11916, January 2023.

[25] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic Chain of Thought Prompting in Large Language Models. 2022. https://arxiv.org/abs/2210.03493.

[26] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-Consistency Improves Chain of Thought Reasoning in Language Models. http://arxiv.org/abs/2203.11171, March 2023.

[27] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. https://arxiv.org/abs/2210.03629v3, October 2022.

[28] Jia Li, Ge Li, Yongming Li, and Zhi Jin. Structured Chain-of-Thought Prompting for Code Generation. May 2023. https://arxiv.org/abs/2305.06599.

[29] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. http://arxiv.org/abs/2305.10601, December 2023.

[30] Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3980–3990, Hong Kong, China, 2019. Association for Computational Linguistics. https://www.aclweb.org/anthology/D19-1410.

[31] Zhongjun Jin, Michael R. Anderson, Michael Cafarella, and H. V. Jagadish. Foofah: Transforming Data By Example. *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 683–698, May 2017. https://dl.acm.org/doi/10.1145/3035918.3064034.

[32] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. Transform-data-by-example (TDE): An extensible search engine for data transformations. *Proc. VLDB Endow.*, 11(10):1165–1177, June 2018. ISSN 2150-8097. https://dl.acm.org/doi/10.14778/3231751.3231766.

[33] Junwen Yang, Yeye He, and Surajit Chaudhuri. Auto-pipeline: Synthesizing complex data pipelines by-target using reinforcement learning and search. *Proc. VLDB Endow.*, 14(11):2563–2575, July 2021. ISSN 2150-8097. https://dl.acm.org/doi/10.14778/3476249.3476303.

[34] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Can Foundation Models Wrangle Your Data? 2022. https://arxiv.org/abs/2205.09911.

[35] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. ReAcTable: Enhancing ReAct for Table Question Answering. 2023. https://arxiv.org/abs/2310.00815.

[36] LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. https://livecodebench.github.io/.

[37] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. http://arxiv.org/abs/2211.11501, November 2022.

[38] Introducing Meta Llama 3: The most capable openly available LLM to date. https://ai.meta.com/blog/meta-llama-3/.

[39] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. https://aclanthology.org/2024.tacl-1.9.

[40] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A Method for Automatic Evaluation of Machine Translation. In Pierre Isabelle, Eugene Charniak, and Dekang Lin, editors, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. https://aclanthology.org/P02-1040.

[41] Chin-Yew Lin. ROUGE: A Package for Automatic Evaluation of Summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. https://aclanthology.org/W04-1013.

[42] Satanjeev Banerjee and Alon Lavie. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare Voss, editors, *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. https://aclanthology.org/W05-0909.

[43] Maja Popović. chrF: Character n-gram F-score for automatic MT evaluation. In Ondřej Bojar, Rajan Chatterjee, Christian Federmann, Barry Haddow, Chris Hokamp, Matthias Huck, Varvara Logacheva, and Pavel Pecina, editors, *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics. https://aclanthology.org/W15-3049.

[44] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. *ArXiv*, September 2020. https://arxiv.org/abs/2009.10297.

[45] Wei Huang, Xudong Ma, Haotong Qin, Xingyu Zheng, Chengtao Lv, Hong Chen, Jie Luo, Xiaojuan Qi, Xianglong Liu, and Michele Magno. How Good Are Low-bit Quantized LLaMA3 Models? An Empirical Study. http://arxiv.org/abs/2404.14047, April 2024.

[46] Mikhail Evtikhiev, Egor Bogomolov, Yaroslav Sokolov, and Timofey Bryksin. Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, 203:111741, September 2023. ISSN 01641212. http://arxiv.org/abs/2208.03133.

[47] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. LoRA Land: 310 Fine-tuned LLMs that Rival GPT-4, A Technical Report. http://arxiv.org/abs/2405.00732, April 2024.

Appendix A

First Appendix

A.0.1 Pandas Method Categories

Table A.1 shows a categorisation of Pandas methods and functions used in data analysis tasks. This was achieved manually by extracting methods used in reference solutions.

Category	**Subcategory**	**Methods/Functions**
Aggregation	Methods	groupby, agg, Grouper
Transformation	Apply	apply, applymap, map
	Reshape	pivot_table, pivot, melt, stack, unstack, transpose
	Bin	cut, qcut
	Explode	explode
	Time Series	shift, pct_change
	Compute	cumsum, diff, rank
Combination	Methods	join, merge, concat, append
Selection	Indexing	loc, iloc, between, filter, isin, isna, isnull, notnull, query, where
	Sorting	sort_values, sort_index
	Subset	nlargest, nsmallest, head, tail, first, last
Cleaning	Missing Data	dropna, fillna, ffill, bfill
	Duplicates	drop_duplicates, duplicated
	Type Conversion	astype, to_numeric, to_datetime, to_period, to_frame, to_list, tolist, ravel
	Renaming	rename, rename_axis
	Structure	reset_index, set_index, reindex, insert
Strings	Methods	str, contains, extract, replace, match, sub, startswith, endswith, split
Computation	Statistics	mean, max, min, median, mode, std, var, quantile, describe
	Aggregation	count, sum, nunique, unique, value_counts
	Correlation	corr, cov
	Arithmetic	add, div, divide, clip, round
Datetime	Conversion	to_datetime, to_timedelta
	Components	dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second
	Operations	strftime, strptime, tz_localize, tz_convert
	Periods	to_period, PeriodIndex, period_range
	Timedelta	Timedelta, timedelta_range
	Offsets	DateOffset, BDay, CDay, Week, MonthEnd, YearEnd
Visualization	Methods	plot, boxplot, hist, lmplot, barplot, scatter, Figure, Layout, Bar, show, Scatter

Table A.1: Pandas and Data Analysis Methods

A.0.2 Sample Prompt

Listing A.1 shows a complete CoT prompt including cell outputs, DataFrames and schema information.

```
You are a genius AI programming data science expert.
Your goal is to output clear detailed steps such that a 5-year old can follow the steps to write Python code.
The current notebook code and markdown cells are provided in between the <notebook_context> tags.
Notebook variables with column information is provided in the <notebook_variables> tags.

<notebook_context>
[showing 24 out of 66 notebook cells]

# In[]:
print(movies_by_month)

# Out[]:
[showing 3 sample rows out of 12 rows]
1     919
5     809
6     827

# In[]:
print(genres_by_popularity)

# Out[]:
[showing 3 sample rows out of 20 rows]
genre
Crime          0.74
Documentary    0.18
Romance        0.59

# In[]:
print(revenueByGeneres)

# Out[]:
[showing 3 sample rows out of 20 rows]
genre
Animation          85256126.91
Science Fiction    86908490.36
Western            46101264.42

# In[]:
print(genresYear2015)

# Out[]:
[showing 3 sample rows out of 19 rows]
Action             107
Science Fiction     86
Western              6

# In[]:
print(genresByYear)

# Out[]:
[showing 3 sample rows out of 56 rows]
release_year
1973    Drama
1978    Drama
2010    Drama

# In[]:
print(vote_count_mean)

# Out[]:
[showing 3 sample rows out of 56 rows]
release_year
1973     94.05
1978     75.35
2010    265.61
```

```

# In[]:
print(rating_by_year)

# Out[]:
[showing 3 sample rows out of 56 rows]
release_year
1973    6.70
1978    6.13
2010    5.99

# In[]:
print(corr)

# Out[]:
[showing 2 sample rows out of 9 rows]
            popularity (float64)  budget (float64)  revenue (float64)  runtime (float64)  vote_count (float64)  vote_average (float64)  release_year (float64)  budget_adj (float64)  revenue_adj (float64)
popularity                  1.00               0.55               0.66    0.14                  0.80                    0.21                    0.09                  0.51                     0.61
budget                      0.55               1.00               0.73    0.19                  0.63                    0.08                    0.12                  0.97                     0.62

# In[]:
print(productionDf)

# Out[]:
[showing 1 sample rows out of 23227 rows]
    popularity (float64)  budget (int64)  revenue (int64)  budget_adj (float64)  revenue_adj (float64)  vote_count (int64)  vote_average (float64)  release_year (int64) release_date (datetime64[ns]) production_companies (str)
0                   1.36        12000000          1456675           11232400.23           1363496.38                  94                     5.6    2013                    2013-09-14             'Gotham Group'

# In[]:
genres_by_popularity.plot(kind = 'bar')
plt.ylabel("Average Popularity")
plt.title("Popularity distribution by genre")
plt.show()

# In[]:
# > We should have expected this. The graph looks pretty similar to the revenue graph. Popularity and revenue should be positively correlated. The more popular a movie is, the more revenue we can expect. Adventure movies have more popularity and consequently have higher revenue.
# - Most Expensive genres

# In[]:
# Average budget per genre?
# To answer this question, we will group our data by genre and find the avg budget.

generesDf.groupby("genre")["budget_adj"].mean().plot(kind = 'bar')
plt.ylabel("Average Budget")
plt.title("Budget distribution by genre")

# In[]:
# > Not suprisingly, Popular and high grossing movies require more budget & vice versa (More the budget, more the marketing, and conseqently more the popularity and revenue):>
# This is also clear from the correlation matrix.
# How are the genres rated on average?

# In[]:
generesDf.groupby("genre")["vote_average"].mean()

# Out[]:
[showing 3 sample rows out of 20 rows]
genre
Crime          6.12
```

```
Documentary     6.91
Romance         6.04

# In[]:
# > Looks like genres doesnt affect the vote_average (Which should be expected ;>)
# <hr>
# Research Question 4
# How often were the movies released in different months?

# In[]:
moviesRaw["monthReleased"] = moviesRaw["release_date"].dt.month
movies_by_month = moviesRaw["monthReleased"].value_counts().sort_index()

# In[]:
print(movies_by_month)

# Out[]:
[showing 3 sample rows out of 12 rows]
1     919
5     809
6     827

# In[]:
movies_by_month.plot(kind = 'bar')
plt.xlabel("Month of the year. 1 for january etc")
plt.title("Feature distribution by month of the movie released")
plt.show()

# In[]:
moviesRaw.groupby("monthReleased").sum()

# In[]:
# > This finding is in conjuction with what is called <b>DUMP MONTHS</b>. See https://en.wikipedia.org/wiki/Dump_months.
# Movies that have high budget/revenue are not released in jan/fed & Aug/Sept since these months are expected to have fewer viewers.
# Research Question 5
# Who are the 5 most casted actors? Return a series with a name and number of movies.

# In[]:
most_casted_5 = castDf['cast'].value_counts().head(5)

# In[]:
most_casted_5.plot(kind = 'bar')
plt.ylabel("Number of movies")
plt.xlabel("Actor")
plt.title("Movie distribution by Actor")

# In[]:
# > Robert De Niro and Samuel L Jackson have the maximum movies!
# Research Question 6

# In[]:
# Production Houses that have the maximum movies!
# For this question, we will use the production houses dataframe!

productionDf.groupby("production_companies")["release_year"].count().sort_values(ascending = False).head().plot(kind = 'bar')
plt.xlabel("Production Houses")
plt.ylabel("Number of movies")
plt.title("Distribution of movies by Production Houses")
</notebook_context>

<notebook_variables>
[some dataframes are not listed due to context limit]

DATAFRAMES: productionDf
COLUMNS: popularity (float64), budget (int64), revenue (int64), budget_adj (float64), revenue_adj (float64), vote_count (int64), vote_average (float64), release_year (int64), release_date (datetime64[ns]), production_companies (str)

DATAFRAMES: corr
```

```
COLUMNS: popularity (float64), budget (float64), revenue (float64), runtime (float64
    ), vote_count (float64), vote_average (float64), release_year (float64),
    budget_adj (float64), revenue_adj (float64)

DATAFRAMES: rating_by_year
COLUMNS: vote_average (float64)

DATAFRAMES: vote_count_mean
COLUMNS: vote_count (float64)

DATAFRAMES: genresByYear
COLUMNS: genre (str)

DATAFRAMES: genresYear2015
COLUMNS: genre (int64)

DATAFRAMES: revenueByGeneres
COLUMNS: revenue_adj (float64)

DATAFRAMES: genres_by_popularity
COLUMNS: popularity (float64)

DATAFRAMES: movies_by_month
COLUMNS: monthReleased (int64)

DATAFRAMES: most_casted_5
COLUMNS: cast (int64)
</notebook_variables>

My question is as follows:
<question>
What are the 5 production houses that generate the most revenue? Return a series
    with a studio name and total revenue.
</question>

For each natural language step, think carefully about the following:
1. All dataframes, series, column names, functions and variables should be in
    quotation marks.
2. Think carefully about column data types. Observe dataframe contents and data
    formats. Consider if parsing is required, such as dates. Ensure null/empty
    values are handled correctly.
3. Think carefully about every column in the dataframe and its relationship with the
     user question.
4. Do not make assumptions. You have been given all context required.

Write your steps in between <scratchpad> tags.
Write your code in between <python> tags.
Do not output any other text.

Take a deep breath and work on this problem step-by-step.
```

Listing A.1: Sample CoT Prompt

A.0.3 Exploratory UI

Figure A.1 shows the Gradio UI for manual analysis of model outputs. This allowed exploration of different error types and model outputs across experiments.

Figure A.1: Gradio UI for analysing results