Automated Extraction of Research Software Installation Instructions from README files: An Initial Analysis

Carlos Utrilla Guerrero[1,2][0000−0002−9994−1462], Oscar Corcho[1][0000−0002−9260−0753], and Daniel Garijo[1][0000−0003−0454−71454]

[1] Ontology Engineering Group, Universidad Politécnica de Madrid
carlos.utrilla.guerrero@alumnos.upm.es, {dgarijo,ocorcho}@upm.es
[2] Research Data and Software (RDS) Department in Delft University of Technology

Abstract. Research Software code projects are typically described with a README files, which often contains the steps to set up, test and run the code contained in them. Installation instructions are written in a human-readable manner and therefore are difficult to interpret by intelligent assistants designed to help other researchers setting up a code repository. In this paper we explore this gap by assessing whether Large Language Models (LLMs) are able to extract installation instruction plans from README files. In particular, we define a methodology to extract alternate installation plans, an evaluation framework to assess the effectiveness of each result and an initial quantitative evaluation based on state of the art LLM models (`llama-2-70b-chat` and `Mixtral-8x7b-Instruct-v0.1`). Our results show that while LLMs are a promising approach for finding installation instructions, they present important limitations when these instructions are not sequential or mandatory.

Keywords: Research/Scientific Knowledge Graphs · Natural Scientific Language Processing · Information Extraction

1 Introduction

Research Software [5] is becoming increasingly recognized as a means to support the results described in scientific publications. Researchers typically document their software project in code repositories, using README files (i.e., `readme.md`) with instructions on how to install, setup and run their software tools. However, software documentation is usually described in natural language, which makes it challenging to automatically verify whether the installation steps required to make the software project work are accurate or not. While seemingly arbitrary, it can be challenging for researchers to follow instructions from different document standards and make sure they work harmonically and consistently.

In this work we aim to address these issues by exploring and assessing the abilities of state of the art Large Language Models (LLMs) to extract installation methods (*Plans*) and their corresponding instructions (*Steps*) from README

files. LLMs such as GPT-4 [21] and MISTRAL [12] have been firmly established as state of the art approaches in various natural scientific language processing (NSLP) tasks related to knowledge extraction from human-like scientific sources such as software documentation from public sharing code hosting services. LLMs have also shown promise in following instructions [26] and learning to use tools [25]. However, existing research in the field is still quite novel.

Our goal in this work is twofold: given a README file, we aim to 1) detect all the available *Plans* (e.g., installation methods for different platforms or operative systems) and, 2) for each Plan, detect what steps are required to install a software project, as annotated by the authors. Our contributions[3] include:

1. *PlanStep*, a methodology to extract structured installation instructions from README files;
2. An evaluation framework to assess the ability of LLMs to capture installation instructions, both in terms of *Plans* and *Steps*;
3. An annotated corpus of 33 research software projects with their respective installation plans and steps.

We implement our approach by following our methodology to evaluate two state of the art LLMs (LLaMA-2 [31] and (MIXTRAL [12]) on both installation instruction tasks with our corpus of annotated projects.

The reminder of the paper is structured as follows. Section 2 discusses relevant efforts to ours, while Section 3 describes our approach. Section 4 describes our experimental setup and early results, Section 5 addresses our limitations and Section 6 concludes the paper.

2 Related work

The goal of extracting relevant information from scientific software documentation forms the foundation of complex knowledge extraction with Natural Language Processing (NLP) models, all of which use machine-learning-based (ML) approaches as basic building blocks [10].

Extracting action sequences from public platforms (e.g. Github, StackOverflow) or README files is an instance of a complex planning tasks class of problems. Remarkably, the field of automated software engineering has rapidly developed novel approaches using LLMs on important problems, for instance, integrating tool documentation [23], detecting action sequence from manuals [18], testing software [39], traceability and generating software specifications [42].

LLMs such as GPT-4 and others follow an architecture using encoder-decoder structure [37], and have been shown to perform well on simple-plans extraction and procedure mining [24], as well as mining to support scientific material discovery [2], [38]. The fundamental constraint of multi-step reasoning abilities, however, remains [35], [33], [19].

[3] The code and corpus are publicly available at: `https://github.com/carlosug/READMEtoP-PLAN/` [44]

In the Knowledge Extraction (KE) field, foundational work builds on general-propose metadata extractor and domain-specific have been successfully applied in a variety of tasks including scientific mentions [6], software metadata extraction [32, 17], and scientific knowledge graph creation [15]. The automated planning community has also continued to push the boundaries of approaches that learn how to extract plans [20] and action sequences from text in domain-specific [13], [4], [8], and general domains (i.e., [18], [35]). Recently, [43], [22] and [26] have made an impressive advance in the feasibility of connecting LLMs with massive APIs documentation. However, in most cases installation instructions, specifically for plans and steps are absent from the corresponding studies.

Recent work [9] [11] has achieved significant improvements in multi-step extraction tasks by using different prompt strategies [36], [34]. In these prompt strategies, the number of operations required to extract entities or events from text grows. This makes it more difficult to learn semantics between the inputs as the instructions are not self-actionable, especially when several steps are involved. Early approaches also discussed the missing descriptions in generated plans [30], and composed learning a mapping from natural language instructions to sequences of actions for planning proposes [27]. In this experiment, this is reduced to a number of formal definitions, albeit at cost of reduced effective resolution due to natural language problem, an effect we plan to counteract with improved prompt variations using formal representation [7, 18] as described in Section 3.6.

To the best of our knowledge, this is the first approach relying entirely on LLMs to extract installation instructions from research software documentation. We focus on eliciting multi-step reasoning by LLMs in a zero-shot configuration. We ask LLMs to extract both the installation plans and step instructions, effectively decomposing the complex task of installing research software into two kinds of sub-problems: 1) extraction of installation methods to capture various ways of installing research software as *Plan*(s) from unstructured documentation, and 2) extraction of installation instructions to identify sequential actions for each method as *Step*(s) (i.e., Step(s) per Plan)

3 ***PlanStep***: Extracting Installation Instructions from README Files

In this section we present *PlanStep*, our proposed approach designed to address limitations briefly outlined in Section 2. First, we describe the core goal and problem we attempt to solve. Next, we describe the *PlanStep* architecture and building blocks. Finally, we describe the data generation and corpus.

3.1 Classical Planning: Software Installation Instructions

The central objective of planning tasks for an intelligent assistant is to autonomously detect the sequence of steps to execute in order to accomplish a

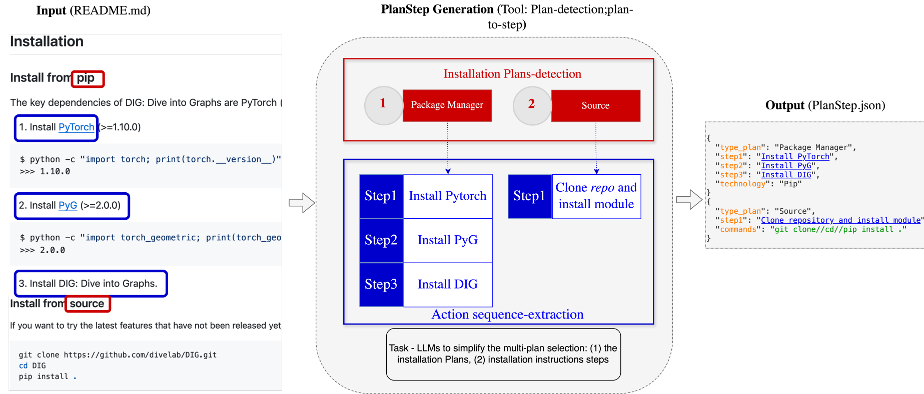

Fig. 1: An example of our experimental approach for *PlanStep*. A research software project includes two installation methods: a simple installation plan *i.e. Package Manager* and a complex one (*i.e. Source*). Each installation method has a different number of steps and configuration details.

task. In classical planning domain, this procedure relies on a formal representation of the planning domain and the problem instance, encompassing actions, and their desired goals [9].

In our case, a problem instance within the installation instruction activity is illustrated in Figure 1. This instance features research software with two alternate plans for installation available in the README: `"Install from pip"` and `"Install from source"`. Each plan is defined fairly briefly, but detailed in the corresponding headers of the markdown file. Subsequently, installation steps outlining the requirements for the setup and execution, are displayed. For instance, the *Plan 1* (categorised as `"Package Manager"`) includes *Steps* contains three steps (or actions). *Plan 2*, classified as *"Source"*, involves one step. If we ask an intelligent assistant to autonomously decide on what sequence of steps to execute this software, we might use an LLM to mine its documentation and break down the installation objective into smaller sub-tasks: first detecting the requirements, then identifying the plan available, and finally execute the necessary commands. Many installation procedures may not need planning. For example, the `"Package Manager"` plan usually entails a one-step with code block, showing exactly what others need to type into their install software from a command line. However, in complex installation plans such as `"from Container"` (i.e., `Docker compose-up`, create virtual environments, configure public keys, etc,..) planning allows the assistant to decide dynamically what steps to take. If we want an assistant to consider a software component and install it following its instructions, the task may be decomposed into different steps: 1) detect alternate plans which are available as installation methods and, 2) for each installation method, detect its corresponding sequence of steps.

3.2 *PlanStep* Methodology

Consistency in the extracted installation methods across different software versions is key for researchers to accurately reproduce experiments, regardless of when and how to access a README file. Therefore, our method aims to consistently connect human-readable instructions to installation *Plans* and *Steps*.

PlanStep receives as input the entire README file, and aims to extract action sequences from text in natural language, representing tasks with two distinct levels of granularity: 1) alternate installation methods (e.g., installation instructions for a separate operative system, installation instructions from source code, etc.), and 2) for each installation method, the sequence of steps associated with it.

Figure 2 depicts the methodology we followed for developing PlanStep, which comprises five stages: the first stage is to collect a set of research software for our study. Second, for each software component in the corpus, we retrieve the link for the code repository, if present, and extract the installation instruction text from its README file. Then, we inspect the original README and represent the alternate installation plans for each README in a structured format. Afterwards, for each entry, we prompt Large Language Models in order to detect plans and their corresponding steps. Finally, we design an evaluation framework to assess the quality of our results.

We limit ourselves to tasks that can be characterized as research software installation activities and involve a reasonable or necessary order of steps to be executed, such as manually setting up a software project component, installing additional libraries using package managers, running from isolated containers, or building from source.

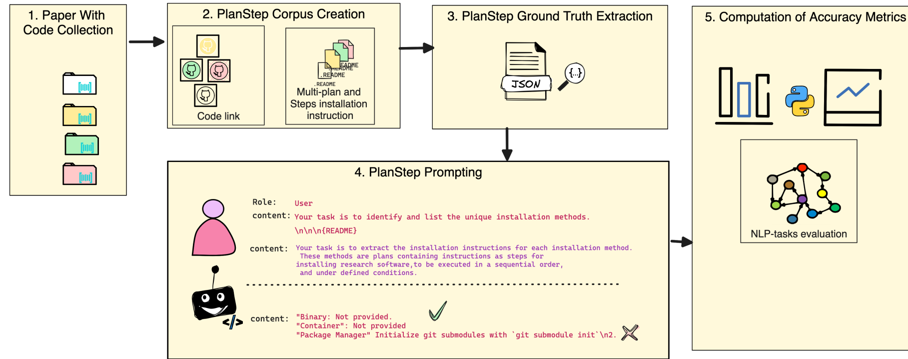

Fig. 2: Overview of the methodology followed to collect research software and design an evaluation framework to assess PlanStep

3.3 *PlanStep* Corpus creation

To systematically evaluate LLM performance on extracting installation plans with steps across varying setups, complexity, and domains, we started by selecting a corpus of research papers with their code[4] implementations from diverse Machine Learning (ML) areas and across different task categories. For this evaluation, we excluded, however, papers with no link to their public repository available on Github or Gitlab.

All annotations were made by the authors of this paper separately, and subsequently compared until consensus was achieved. We discussed each entry to determine the final set of steps and plans for each research software. Very rarely, agreement on specific properties remained elusive even after evaluation, and these cases were manually resolved through additional discussion. In summary, our corpus has 33 fully active and maintained open-source research software projects.

3.4 Ground Truth Extraction for *PlanStep*

The 33 research software projects in our corpus were selected as study subjects. In a manual co-annotation process, we tasked annotators with identifying both the installation plans and steps associated with each project's README. The installation plans varied in complexity and description style, with some, like `from pip` typically comprising a single-sentence step (excluding requirements), while others, such as 'from source,' included multiple steps, considering various user environments and requirements. Additionally, we defined specific properties for each plan type, taking into account technology-specific support, such as package repositories like `npm` or `PyPI`. Further elaboration on these definitions is provided below:

A. Plan: represents the concept of an installation method available in a README, which is composed of steps, that must be executed in a given order. For instance, a "Source" is an instance of a Plan concept. A README can include one or multiple Plans in the installation instructions section. A brief explanation of the plans and examples is provided in Table 1.

B. Step: represents the concept of a planned action as part of a '*Plan*' to be executed sequentially. It may consist of either a single action or a group of actions. We define a 'Step' based on the original README text, where consecutive actions mentioned together are annotated as one step. For instance, Listing 1.4 illustrates this concept with a simple JSON example. In the example, the authors' original text describes the first step (`Step1`) as `'Clone this repository and install requirements,'` which encompasses two distinct actions: `'Clone`

[4] we made use of Paper with Code platform: https://paperswithcode.com/ as it links together articles with software repository

Plan Type	**Short definition**	**Examples**
Binary	Installing by directly downloading and running precompiled files.	GitHub releases
Container	Installing by packaging software components and its dependencies using containerization.	Docker, Podman, Singularity
Package Manager	Downloading and installing from official repositories.	Conda, Homebrew, Pip, npm
Source	Installing by manually compiling original code into machine-readable binaries.	Clone from repository, create virtual environment

Table 1: Definition of plan types and examples found in our corpus

`this repository’` and `‘install requirements.‘`. The second step (`Step2`) simply involves one action `‘iRun the container with docker - compose’`

```
{
 "id": "4",
 "name": "utiasASRL/steam_icp",
 "url": "https://raw.githubusercontent.com/utiasASRL/steam_icp/master/README.
    md",
 "plans": [
  {
   "type": "Container",
   "steps": [
    {
     "text": "Clone this repository and install requirements.",
     "seq_order": 1,
     "is_optional": false
    },
    {
     "text": "Run the container with docker-compose",
     "seq_order": 2,
     "is_optional": false
    }
 ],
 "README_instructions": "## Installation Clone this repository and its
    submodules.[....]"
}
```

Listing 1.1: JSON snippet showing Step 1 including two actions (Steps) *“clone”* and *“install”*

We manually examined cases where annotators disagreed. For example, significant confusion arose from overly complicated instructions detected in README files, particularly in cases where installation instructions were included in the markdown subheadings, such as `#Step1: Download the files`, followed by a paragraph like `Step1: Download the files with the following commands`. We resolved these conflicts by removing the content of these subheadings and providing detailed annotations for the subsequent paragraph.

Next, we faced challenges when describing plan types and steps across supported technologies. For instance, while instructions for the package manager plan typically involve running pip install, the `TorchCP` library offered alternative installation methods like the `TestPyPI` server. To resolve this, we created a distinct plan named ”package manager” and specified `TestPyPI` as the associated technology property.

Lastly, conflicts emerged concerning the inclusion of installation requirements. Some cases listed requirements within the installation instructions, while others deposited them in a separate section, traditionally before the installation instruction content begins. We decided to include these software requirements specifications only when they were part of the installation instruction section content.

3.5 Distribution of the installation instructions of README files

Table 2 shows descriptive statistics of the selected research software projects based on our annotations. We reported four distinct installation Plans: binary, source, package manager and container. Notably, over half of our corpus exclusively relied on the "source" method for installing research software via README files. While "from source" was the most prevalent standalone method (66%), container and package manager plans were observed in only two and one cases, respectively. As anticipated, the "binary" method was not reported at all, indicating its rarity on open source general repositories such as Github. Unsurprisingly, the most popular research software tools e.g., `tensorflow` or `langchain` incorporated the instructions to install with package manager, typically consisting of up to two steps. The Plans vary widely in their number of steps. For example, *"simple"* Plans e.g., Package Manager and Container consists of 2-3 steps, while *"complex"* featured 10 (see Table 2 col (Total Steps)). This diversity in the number of steps impacts installation Plan length in two ways: 1) more steps introduce more complexity, and 2) additional instructions can serve as obstacles, needing further action for installation.

Approximately 44% of our samples offered multiple plans or combinations for software installation, suggesting a diverse landscape of installation approaches. Further analysis of these combinations revealed redundant information across many instruction sections, highlighting potential challenges for LLMs in accurately identifying plans and steps. For instance, the maximum length of installation instructions for the source plan reached approximately 1,765 tokens, underscoring the complexity and variability of these instructions. This diversity not only reflects the varied nature of installation plans but also poses challenges for LLMs in accurately parsing and selecting relevant instructions, potentially leading to errors in plan and step detection. The total average length of installation instructions across all subjects was 130.79 tokens.[5]

3.6 *PlanStep* prompting

This section introduces our PlanStep prompt templates and its explanations for such tasks, as depicted in PROMPT101 and PROMPT201.

[5] Additional details about the corpus and its data exploration are available in our GitHub repository: `https://github.com/carlosug/READMEtoP-PLAN/blob/main/RESULTS/corpus-explore-data.ipynb`

Plan Types(*Combinations*)	**Counts**(Prop. %)	Max. **Total Steps**	Max.Tokens$_{\text{Install_md}}$
Source	22 (66.66%)	10	253
Container	2 (6.06%)	3	158
Package Manager	1 (3.03%)	2	102
Binary	- (-%)	-	-
Mult. Binary	1 (3.03%)	1	108
Mult. Source	1 (3.03%)	3	303
Mult. Package Manager	3 (9.09%)	3	348
Package Manager & Source	1 (3.03%)	3	217
Package Manager & Binary	1 (3.03%)	2	187
Container & Source	1 (3.03%)	9	725
ALL	33 (100%)		

Table 2: Statistics of plan and steps in the corpus. We report the number and average of "ids" per plan type, multiple plans, maximum total steps in a plan, and the length of installation instructions with parameters (TokenInstall.)

We directly instruct the LLMs with prompt design to describe the installation methods (Plan) and their corresponding installation instructions (Step) for each README. That is, the usual zero-shot prompt is set to ask LLM two tasks, Plan and Step, respectively. Since the LLM contains no information about these terms, we describe the terms and their respective meanings next to the task of the prompt. Consequently, the prompts used in our experiment can be categorised as follows:

Plan prompting: This task is about extracting the installation method as Plans described in a README. We named it the PROMPT101, and it contains the four unique Plans and its definitions.

Step prompting: This task asks for detecting the installation instructions as Steps found in a README. We named it the PROMPT201 and it requests a list of Steps for a given installation plan.

4 Experiments

In order to evaluate the effectiveness of our approach, we conducted experiments to test the ability of LLMs to capture plans and the sequence of tasks required to install different software.

4.1 Experimental Setup

We employed `Mixtral-8x7b-Instruct-v0.1` [12] and `LLaMA2-70b-chat` [31], which are two of the most widely-used open-source LLMs with public access. [6] Both models demonstrate moderately good instruction-following capabilities [43]. Throughout our experiments, we maintained a `temperature` of 0 (argmax sampling) to ensure reproducible results. The ground-truth annotations and study

[6] Accessible through a public Python API Library: https://pypi.org/project/groq/

Plan Task (PROMPT101)

Plan Task (PROMPT101):
Given the following README, your task is to identify and list the unique installation methods. These methods are plans containing instructions for installing research software, to be executed in a specific order and under defined conditions. Exclude code commands. Be concise.

1. Binary: Install via download and run precompiled files. For example, GitHub releases.
2. Container: Install the software and its dependencies via isolated environments. For example, Docker, Podman, or Singularity.
3. Package Manager: Install via tools and indexed repositories. For example, Conda, Homebrew, or Pip.
4. Source: Run via command-line, manage and install dependencies, compile source code to a target machine, build, and run. For example, download raw source code, clone repositories, and install dependencies from code repositories.

Step Task (PROMPT201)

Step Task (PROMPT201):
Given the following README, extract the installation instructions for each installation method. These methods are plans containing instructions as steps for installing research software, to be executed in a sequential order, and under defined conditions. Exclude code commands. Be concise.

1. Binary:[....] 4. Source[a].

[a] We insert the same definitions as states in PROMPT101

LLM	Zero-shot		
	Precision	Recall	F1 score
`llama-2-7b-chat`	0.4615	0.8333	**0.5941**
`Mixtral-8x7b-Instruct-v0.1`	0.4068	0.6667	0.5053

Table 3: Results obtained on the Plan detection task. The best result is **boldfaced**.

subjects used to compare LLM's predicted responses in the experiments were those presented in Section 3.3 and Section 3.4.

4.2 Evaluation Metrics

To assess our proposed *PlanStep* method, we employed the following metrics to assess the performance of LLMs on NLP-oriented tasks (as proposed by [3]):

- **F1-scores**: these scores are computed to compare the performance of LLMs in extracting plans with the ground truth annotations.
- **Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [16]**: we report ROUGE-1 (R1), ROUGE-2 (R2), and ROUGE-L to evaluate the quality of of the results by comparing the LLMs extracted steps with the ground-truth dataset.

4.3 Evaluation Results

The results of our evaluation are shown in Table 3 and Table 4 for Plan and Step tasks respectively. We present the performance of open-source LLM on two task with the standard (PROMPT101 and PROMPT201) zero-shot prompt templates.

Plan-task: To evaluate the effectiveness of the LLMs, we tested the models in different ways, measuring the change in performance on plan task by comparing their generated response plans with ground truth annotations. We used zero-shot approach. Table 3 summarises our results on plan tasks, and compares both LLMs' performance.

Both LLMs in zero-shot prompting achieved roughly a performance of more than 50% F1-score. LLaMA-2 exhibits superior performance over MIXTRAL in plan task with the LLaMA-2 outperforming the best of our experiment by 9% compared to MIXTRAL.

Step-task: We evaluated the performance with three metrics to measure the quality in analyzing step-task. Table 4 shows the performance of the models (e.g., MIXTRAL and LlaMA).

LLM	Zero-shot		
	R1 ↑	R2 ↑	RL
`llama-2-7b-chat`	29.48	18.88	27.75
`Mixtral-8x7b-Instruct-v0.1`	46.42	37.53	45.27

Table 4: Evaluation results for detecting task steps for each plan. The scores (%) for Rouge-1 (R1), Rouge-2 (R2), and Rouge-L (RL) for the generated step descriptions compare our results against the ground truth steps.

We further observe that while MIXTRAL consistently outperforms LLaMA-2 across all ROUGE scores (R1, R2, and RL) in the *Step-task*, achieving approximately 15% higher scores, both models demonstrate similarly poor performance in adhering to optimal step orderings, with scores ranging from 0.46 to 0.29. These findings suggest that both models struggle with the task of sequentially ordering steps in a installation Plan.

4.4 Analysis

Results of Plan-Step task. Experimental results indicate that both LLMs scored an average of around 55% F1-score for plan-task, and 37% ROUGE scores for step-task. This suggest that LLMs intrinsically vary in their abilities to solve complex tasks and reason efficiently, which are crucial for extracting plans and detecting steps more accurately.

Error Analysis. We performed a detailed analysis on specific cases where the detection performance of the LLMs differ significantly from the annotations to understand why certain steps were falsely detected. We manually studied all errors made by LLMs and classify them into four categories. Table 5[7] shows the count of each error type on Plan-Step tasks: **E1:** means models call Plans and Steps installation instructions wrongly by reusing prompt input `e.g.,"Binary": ["Step 1: Definition Prompt."`; **E2:** indicates cases where models include notes and code commands in their responses, resulting in falsely imputed new steps to a wrong Plan; **E3** refers to situations where models extract steps correctly but assign them to the wrong Plan Types due to a mixture of verbs or words associated with the different methods, and lack of context e.g., if the word `"pip"` appears, the LLM directly assigns the corresponding step to directly "Package Manager"[8]; **O** represents errors in an unclassified category (e.g., summarizing steps, incorporating steps from a previous README, splits steps or invented sentences as hallucinations). Further tables, plots and error responses examples can be found in the Appendix.

[7] the raw material we used to calculate the counts are listed in our repository [44]: `qualitative_error_analysis.md`

[8] install python packages from a git repository has been classified as step of "Source" plan"

Our results suggest that different Plans exhibit a wide diversity in error types: simple installation tasks with few actions (“Package” and “Binary”) primarily encounter issues related to **E3**; notably, “Source” faces more issues with **E1**, indicating a significant impact of prompts on model performance. Across Plan types, we observe nearly identical results suggesting a possible explanation: concise instructions in README files may significantly reduce these incorrect behaviors, leading to successful execution of installation steps. Additional experiments are needed to assess this hypothesis.

	LLAMA				**MISTRAL**			
Error Type:	E1	E2	E3	O	E1	E2	E3	O
Binary	0	0	1	0	0	0	0	1
Source	15	4	8	1	0	6	4	2
Package Manager	0	0	1	0	1	0	4	0
Container	0	1	1	1	0	0	1	0

Table 5: Counts of *PlanStep* on different Plans. E1: wrong Plans category but correct Steps; E2: wrong order of steps but correct number; E3: wrong sequential order; E4: unanswerable with API; O: others

Effect of Prompts. Figure 3 shows an overview of the steps detected by each LLM. Both MIXTRAL and LLaMA-2 perform on par *(score: 10 vs. 7)* in detecting steps correctly and incorrectly (score: 8 vs. 9). However, the latter exhibits slightly worse performance compared to the former in over-detection cases *(score: 4 vs. 9)*, which is likely due to the ability of the model to insert prompts information into the responses. The definitions seem to inadvertently lead LLMs to incorrectly detect plans and steps by copying extra steps added solely to the prompt definition i.e., **E1:** call non-existing Plans by adding prompt’s input. This observation suggests the need for additional testing with zero-shot prompts for different installation plans (and reduce the definitions used in the prompt). More advanced zero-shot prompting methods [40] as well as chain-of-thought prompt strategies [41] to effectively guide LLMs in translating steps into smaller sub-tasks will be investigated in our future work.

Few-shot prompt strategies such as LLM4PDDL [28] together with chain-of-thought prompts, may provide an expressive and extensible vocabulary representation for semantically writing and describing plans to machines. We plan to investigate this approach further in future work.

5 Discussion

This work aims to automatically extract all available installation information from research software documentation. Our experiments demonstrate that while

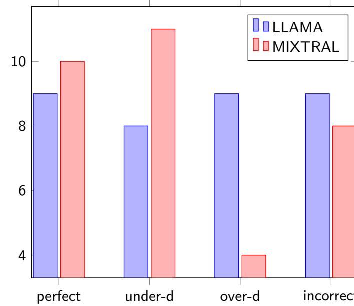

Fig. 3: Total count of steps detected for each Plan per LLM, in comparison with the ground truth. If a LLM detected fewer steps than the annotations, we consider it under-detection **(under-d)**, while if it detected more, it indicate over-detection **(over-d)**. A correct step detection (**(perfect)** indicates the number of steps agree with those in the ground truth. The **(incorrect)** detection counts steps in a plan that are falsely detected i.e., the LLM model detected a plan with steps that are not part of our annotations)

LLMs show promising results, there is substantial room for improvement. During our analysis, we prioritized extracting concise plans and steps of software installation text using two LLMs. LlaMA-2 generally demonstrates the fewest errors in plan-task, indicating a higher accuracy in predicting the installation methods. The LlaMA-2, however,shows a progressively higher number of errors when dealing with steps. MIXTRAL exhibits the opposite. We observe that MIXTRAL outputs are significantly more truthful than LlaMA-2 with less randomness and creativity in their responses. Notably, the more steps involved, the more frequent errors across both models, indicating the challenges faced in accurately predicting parameters for tools.

Moreover, the reliance on LLM for the evaluation of plans and step instructions introduce new challenges. As LLM's ability in planning tasks in under scientific scrutiny [29], [14], there is a crucial need for further validation and fine-tuning of its capabilities in this specific context.

We are in our initial phase of the experimental research project, and consequently, components from *PlanStep* approach will certainly be updated and revised. First, we believe that designing combinations of few-shot prompt standards with the addition of strict formal language will improve the ability of LLMs to detect plans, and their instructions for installation consistently. Second, additional evaluations are needed to validate the insights obtained in our experiments. For plan tasks, our approach may be compared with baseline models, measuring the change in performance. Third, increasing the size of our annotated corpus is notably advantageous, providing a broader exploration of alternative semantic approaches formal representations. However, the manual nature of our instruction writing process limits our capacity to scale this work significantly.

6 Conclusion and Future Work

In this work we presented an evaluation framework and initial experimentation for using LLMs as a means to extract alternate research software installation plans and their corresponding instructions. Our approach involves equipping the LLM with essential documentation tailored to installation instructions, enabling them to refine their accuracy when using the README and improve their performance in automating the detection of installation instructions. As part of our evaluation framework we have proposed an annotated corpus, which collects different research software with their installation instructions, to systematically evaluate LLMs in extracting tasks, including plans and steps belonging to those plans.

Our experiments show promising results for both plan detection and step detection, although we are still a long way from our goal. We are currently extending our approach in different directions. First, we are augmenting the annotation corpus to consider additional README files of increasing complexity in order to create a comprehensive benchmark, distinguishing READMES of different complexity. Second, we aim to improve the prompting strategies used in our approach, including few-shot examples to better equip the model with the goal of each PlanStep task. Our central goal is to create an assistant that aids in installing research software while addressing issues that may currently exist in the installation instructions. Investigating further the addition of executable instructions in formalised and machine-readable language from classical planning research community *i.e., Domain Definition Language (PDDL)* [1] and beyond i.e., *P-Plan Ontology* [7] is another research goals of ours.

Acknowledgements

This work is supported by the Madrid Government (Comunidad de Madrid - Spain) under the Multiannual Agreement with Universidad Politécnica de Madrid in the line Support for R&D projects for Beatriz Galindo researchers, in the context of the VPRICIT, and through the call Research Grants for Young Investigators from Universidad Politécnica de Madrid. The authors would also like to acknowledge European Union's Horizon Europe Programme under GA 101129744 — EVERSE — HORIZON-INFRA-2023-EOSC-01-02.

References

[1] Constructions Aeronautiques et al. "Pddl— the planning domain definition language". In: *Technical Report, Tech. Rep.* (1998).

[2] Microsoft Research AI4Science and Microsoft Azure Quantum. "The impact of large language models on scientific discovery: a preliminary study using gpt-4". In: *arXiv:2311.07361* (2023).

[3] Kathrin Blagec et al. "A global analysis of metrics used for measuring performance in natural language processing". In: *arXiv:2204.11574* (2022).

[4] Daniil A. Boiko et al. “Autonomous chemical research with large language models”. In: *Nature* 624.7992 (Dec. 2023). Number: 7992 Publisher: Nature Publishing Group, pp. 570–578. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06792-0. URL: https://www.nature.com/articles/s41586-023-06792-0 (visited on 12/31/2023).

[5] Neil P. Chue Hong et al. *FAIR Principles for Research Software (FAIR4RS Principles)*. Version 1.0. June 2022. DOI: 10.15497/RDA00068.

[6] Caifan Du et al. “Softcite dataset: A dataset of software mentions in biomedical and economic research publications”. In: *Journal of the Association for Information Science and Technology* 72.7 (2021), 870–884. ISSN: 2330-1635. DOI: 10.1002/asi.24454.

[7] Daniel Garijo and Yolanda Gil. “Augmenting PROV with Plans in P-PLAN: Scientific Processes as Linked Data”. In: *Second International Workshop on Linked Science: Tackling Big Data (LISC), held in conjunction with the International Semantic Web Conference (ISWC)*. Boston, MA, 2012.

[8] Eran Hirsch, Guy Uziel, and Ateret Anaby-Tavor. “What’s the Plan? Evaluating and Developing Planning-Aware Techniques for LLMs”. In: *arXiv:2402.11489* (2024).

[9] Eran Hirsch, Guy Uziel, and Ateret Anaby-Tavor. *What’s the Plan? Evaluating and Developing Planning-Aware Techniques for LLMs*. Feb. 18, 2024. arXiv: 2402.11489[cs]. URL: http://arxiv.org/abs/2402.11489 (visited on 03/14/2024).

[10] Xinyi Hou et al. *Large Language Models for Software Engineering: A Systematic Literature Review*. Aug. 2023. URL: http://arxiv.org/abs/2308.10620 (visited on 09/05/2023).

[11] Xu Huang et al. “Understanding the planning of LLM agents: A survey”. In: *arXiv:2402.02716* (2024).

[12] Albert Q Jiang et al. “Mixtral of experts”. In: *arXiv:2401.04088* (2024).

[13] Qiao Jin et al. *GeneGPT: Augmenting Large Language Models with Domain Tools for Improved Access to Biomedical Information*. May 16, 2023. arXiv: 2304.09667[cs,q-bio]. URL: http://arxiv.org/abs/2304.09667 (visited on 03/14/2024).

[14] Subbarao Kambhampati et al. “LLMs Can’t Plan, But Can Help Planning in LLM-Modulo Frameworks”. In: *arXiv:2402.01817* (2024).

[15] Aidan Kelley and Daniel Garijo. “A Framework for Creating Knowledge Graphs of Scientific Software Metadata”. In: *Quantitative Science Studies* (Nov. 2021), pp. 1–37. ISSN: 2641-3337. DOI: 10.1162/qss_a_00167.

[16] Chin-Yew Lin. “ROUGE: A Package for Automatic Evaluation of Summaries”. In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://www.aclweb.org/anthology/W04-1013.

[17] A. Mao, D. Garijo, and S. Fakhraei. “SoMEF: A Framework for Capturing Scientific Software Metadata from its Documentation”. In: *2019 IEEE*

International Conference on Big Data (Big Data). 2019, pp. 3032–3037. DOI: 10.1109/BigData47090.2019.9006447.

[18] Shivam Miglani and Neil Yorke-Smith. “NLtoPDDL: One-Shot Learning of PDDL Models from Natural Language Process Manuals”. In: *ICAPS’20 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS’20)* (2020).

[19] Philipp Mondorf and Barbara Plank. “Beyond Accuracy: Evaluating the Reasoning Behavior of Large Language Models–A Survey”. In: *arXiv:2404.01869* (2024).

[20] Alberto Olmo, Sarath Sreedharan, and Subbarao Kambhampati. *GPT3-to-plan: Extracting plans from text using GPT-3*. June 13, 2021. arXiv: 2106.07131[cs]. URL: http://arxiv.org/abs/2106.07131 (visited on 01/17/2024).

[21] OpenAI. *GPT-4 Technical Report*. Mar. 27, 2023. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774 (visited on 09/24/2023).

[22] Yiwei Qin et al. *InFoBench: Evaluating Instruction Following Ability in Large Language Models*. Jan. 7, 2024. arXiv: 2401.03601[cs]. URL: http://arxiv.org/abs/2401.03601 (visited on 02/16/2024).

[23] Yujia Qin et al. *ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs*. Oct. 3, 2023. arXiv: 2307.16789[cs]. URL: http://arxiv.org/abs/2307.16789 (visited on 02/16/2024).

[24] Anisa Rula and Jennifer D’Souza. “Procedural Text Mining with Large Language Models”. In: *Proceedings of the 12th Knowledge Capture Conference 2023*. K-CAP ’23. New York, NY, USA: Association for Computing Machinery, Dec. 5, 2023, pp. 9–16. DOI: 10.1145/3587259.3627572.

[25] Timo Schick et al. *Toolformer: Language Models Can Teach Themselves to Use Tools*. Feb. 9, 2023. arXiv: 2302.04761[cs]. URL: http://arxiv.org/abs/2302.04761 (visited on 09/21/2023).

[26] Yongliang Shen et al. *TaskBench: Benchmarking Large Language Models for Task Automation*. Dec. 9, 2023. arXiv: 2311.18760[cs]. URL: http://arxiv.org/abs/2311.18760 (visited on 03/14/2024).

[27] Mohit Shridhar et al. *ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks*. Mar. 30, 2020. arXiv: 1912.01734[cs]. URL: http://arxiv.org/abs/1912.01734 (visited on 01/16/2024).

[28] Tom Silver et al. “PDDL planning with pretrained large language models”. In: *NeurIPS 2022 foundation models for decision making workshop*. 2022.

[29] Kaya Stechly, Karthik Valmeekam, and Subbarao Kambhampati. “On the Self-Verification Limitations of Large Language Models on Reasoning and Planning Tasks”. In: *arXiv:2402.08115* (2024).

[30] Moritz Tenorth, Daniel Nyga, and Michael Beetz. “Understanding and executing instructions for everyday manipulation tasks from the World Wide Web”. In: *2010 IEEE International Conference on Robotics and Automation*. 2010 IEEE International Conference on Robotics and Automation (ICRA 2010). Anchorage, AK: IEEE, May 2010, pp. 1486–1491.

ISBN: 978-1-4244-5038-1. DOI: 10.1109/ROBOT.2010.5509955. (Visited on 02/02/2024).

[31] Hugo Touvron et al. "Llama 2: Open foundation and fine-tuned chat models". In: *arXiv:2307.09288* (2023).

[32] Jason Tsay et al. "AIMMX: Artificial Intelligence Model Metadata Extractor". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. New York, NY, USA: Association for Computing Machinery, Sept. 18, 2020, pp. 81–92. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387448. (Visited on 09/20/2023).

[33] Karthik Valmeekam et al. "Large Language Models Still Can't Plan (A Benchmark for LLMs on Planning and Reasoning about Change)". In: ().

[34] Karthik Valmeekam et al. "On the planning abilities of large language models-a critical investigation". In: *Advances in Neural Information Processing Systems* 36 (2024).

[35] Karthik Valmeekam et al. *PlanBench: An Extensible Benchmark for Evaluating Large Language Models on Planning and Reasoning about Change*. Nov. 25, 2023. arXiv: 2206.10498[cs]. URL: http://arxiv.org/abs/2206.10498 (visited on 01/18/2024).

[36] Karthik Valmeekam et al. "Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change". In: *Advances in Neural Information Processing Systems* 36 (2024).

[37] Ashish Vaswani et al. "Attention is All you Need". In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017.

[38] Hanchen Wang et al. "Scientific discovery in the age of artificial intelligence". In: *Nature* 620.7972 (Aug. 2023). Number: 7972 Publisher: Nature Publishing Group, pp. 47–60. ISSN: 1476-4687. DOI: 10.1038/s41586-023-06221-2. (Visited on 09/07/2023).

[39] Junjie Wang et al. "Software testing with large language models: Survey, landscape, and vision". In: *IEEE Transactions on Software Engineering* (2024).

[40] Lei Wang et al. *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models*. May 26, 2023. arXiv: 2305.04091[cs]. URL: http://arxiv.org/abs/2305.04091 (visited on 03/14/2024).

[41] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Advances in neural information processing systems* 35 (2022), pp. 24824–24837.

[42] Danning Xie et al. *Impact of Large Language Models on Generating Software Specifications*. June 5, 2023. arXiv: 2306.03324[cs]. URL: http://arxiv.org/abs/2306.03324 (visited on 09/11/2023).

[43] Siyu Yuan et al. *EASYTOOL: Enhancing LLM-based Agents with Concise Tool Instruction*. Jan. 11, 2024. arXiv: 2401.06201[cs]. URL: http://arxiv.org/abs/2401.06201 (visited on 02/06/2024).

[44] Carlos Z. *carlosug/READMEtoP-PLAN: READMEtoP-PLAN First Release*. Apr. 2024. DOI: 10.5281/zenodo.10991890.

Appendix

Examples of Error types produced by LLMs:

```
1  Question: PROMPT201
2  Correct Answer:
3      },
4      "7": {
5          "Source": [
6              "### Dependencies \n Initialize git submodules with",
7              "### Python environment \n Install the specific versions of every
       package from `requirements.txt` in a new conda environment:",
8              "To ensure that Python paths are properly defined, update the
      `~/.bashrc` by adding the following lines"
9          ]
10     },
11 LLAMA Answer:
12     "7": {
13         "Binary": [
14             "Step 1: Download the precompiled files from GitHub releases."
15         ],
16         "Container": [
17             "Step 1: Initialize git submodules with `git submodule init` and
      `git submodule update`.",
18             "Step 2: Create a new Conda environment with the specific
      versions of every package from `requirements.txt` using `conda create --
      name gsft python=3.9`.",
19             "Step 3: Activate the environment using `conda activate gsft`.",
20             "Step 4: Install the packages listed in `requirements.txt` using
      `pip install -r requirements.txt`."
21         ],
22         "Source": [
23             "Step 1: Clone the repository.",
24             "Step 2: Initialize git submodules with `git submodule init` and
      `git submodule update`.",
25             "Step 3: Create a new Conda environment with the specific
      versions of every package from `requirements.txt` using `conda create --
      name gsft python=3.9`.",
26             "Step 4: Activate the environment using `conda activate gsft`.",
27             "Step 5: Install the packages listed in `requirements.txt` using
      `pip install -r requirements.txt`."
28         ]
29     },
```

Listing 1.2: Demonstration id: 7 (Error Type. 1 - Adding definition in the LLM answer)

```
1  Question: PROMPT201
2  Correct Answer:
3      "14": {
4          "Source": [
5              "Install the requirements with:",
6              "or set up TPU VM with:"
7          ]
8      },
9  MISTRAL Answer:
10     "14": {
11         "Package Manager": [
12             "1. Create a new conda environment with Python 3.10.",
13             "2. Activate the new conda environment.",
14             "3. Install Jax with CUDA 12 support: `pip install -U `jax[
      cuda12_pip]==0.4.23` -f https://storage.googleapis.com/jax-releases/
      jax_cuda_releases.html`",
15             "4. Install the required packages: `pip install -r requirements.
      txt`"
16         ]
17     },
```

Listing 1.3: Demonstration id: 14 - Error Type. 2 (Mixture of steps from code commands and notes)

```
1 Question: PROMPT201
2 Correct Answer:
3     },
4     "13": {
5         "Container": [
6             "First, install Docker on your machine. Then move to the original
       directory `learning_to_fly` and build the Docker image:",
7             "If desired you can also build the container for building the
      firmware:",
8             "After that you can run it using e.g.:",
9             "Navigate to [https://0.0.0.0:8000](https://0.0.0.0:8000) with
      your browser, and you should see something like in the screenshot above (
      after starting the training).",
10            "Navigate to [https://0.0.0.0:6006](https://0.0.0.0:6006) with
      your browser to investigate the Tensorboard logs."
11        ],
12        "Source": [
13            "Clone this repository:",
14            "Then instantiate the `RLtools` submodule:",
15            "Then instantiate some dependencies of `RLtools` (for
      conveniences like checkpointing, Tensorboard logging, testing, etc.):",
16            "#### Install dependencies on Ubuntu",
17            "#### Install dependencies on macOS",
18            "Going back to the main directory (`learning_to_fly`), we can now
       configure the build of the code:",
19            "Finally, we can build the targets:",
20            "After successfully building the targets, we can run the code (in
       the original directory `learning_to_fly`):",
21            "If not already installed:",
22            "Then from the original directory `learning_to_fly`:",
23            "To run the training with the UI, we download the JavaScript
      dependencies in the form of the two files `three.module.js` and `
      OrbitControls.js`:",
24            "After that we can execute the UI binary from the root folder:",
25            "Now you should be able to navigate to [http://0.0.0.0:8000](http
      ://0.0.0.0:8000) in your browser and start the training."
26        ]
27    },
28 LLAMA Answer:
29    "13": {
30        "Container": [
31            "Step 1: Install Docker on your machine.",
32            "Step 2: Move to the original directory `learning_to_fly` and
      build the Docker image: `docker build -t arpllab/learning_to_fly .`",
33            "Step 3: Run the container using `docker run -it --rm -p
      8000:8000 arpllab/learning_to_fly`.",
34            "Step 4: Navigate to [https://0.0.0.0:8000](https://0.0.0.0:8000)
       with your browser to access the training program."
35        ],
36        "Source": [
37            "Step 1: Clone the repository: `git clone https://github.com/
      arplaboratory/learning-to-fly`.",
38            "Step 2: Instantiate the `RLtools` submodule: `git submodule
      update --init -- external/rl_tools`.",
39            "Step 3: Instantiate some dependencies of `RLtools`: `git
      submodule update --init -- external/cli1 external/highfive external/json/
       external/tensorboard tests/lib/googletest/`.",
40            "Step 4: Install dependencies on Ubuntu or macOS.",
41            "Step 5: Configure the build: `cmake .. -DCMAKE_BUILD_TYPE=
      Release -DRL_TOOLS_BACKEND_ENABLE_OPENBLAS:BOOL=ON` (for Ubuntu +
      OpenBLAS) or `cmake .. -DCMAKE_BUILD_TYPE=Release -
      DRL_TOOLS_BACKEND_ENABLE_MKL:BOOL=ON` (for Ubuntu + MKL) or `cmake .. -
      DCMAKE_BUILD_TYPE=Release` (for macOS).",
42            "Step 6: Build the targets: `cmake --build . -j8`.",
43            "Step 7: Run the code: `cd .. ./build/src/training_headless`.",
44            "Step 8: To run the training with the UI, download the JavaScript
       dependencies and execute the UI binary: `cd src/ui ./get_dependencies.sh
      ` and `cd ../../ ./build/src/ui 0.0.0.0 8000`.",
45            "Step 9: Navigate to [http://0.0.0.0:8000](http://0.0.0.0:8000)
      in your browser to access the training program."
46        ]
47    },
```

Listing 1.4: Demonstration id: 13 - Error Type. O (correct sequential order but llm split and invented sentences)

Detailed Tables:

– Figure 4 illustrates the distribution of total length of readme instructions for each study subject in our corpus.
– Figure 5 aggregates the study subjects per distinct Plan type and its technology properties.

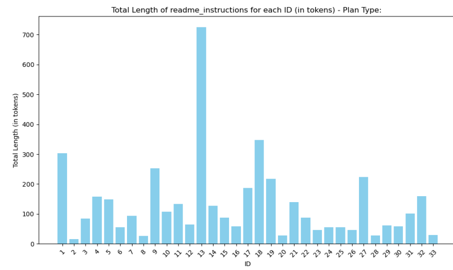

Fig. 4: Length (Tokens) of the README for each ”id” research software

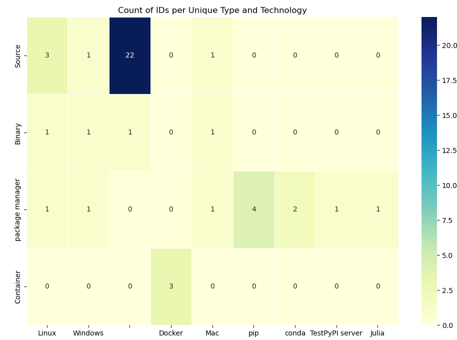

Fig. 5: Heatmap of our corpus

– Plots where each bar represents an ID research software, and within each bar, different colored segments represent the ratio of system-detected steps to reference steps for each method . Ratio of LLM Detected steps to Annotations steps. A value around 1 indicates a good match between LLM and Annotations

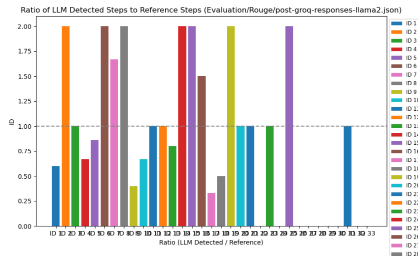

Fig. 6: LLM LLAMA2

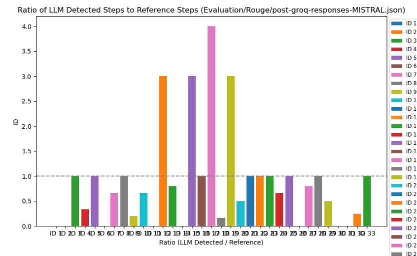

Fig. 7: LLM MIXTRAL