Beyond pip install: Evaluating LLM Agents for the Automated Installation of Python Projects

Louis Milliken
School of Computing
KAIST
Daejeon, Republic of Korea
lmilliken@kaist.ac.kr

Sungmin Kang
School of Computing
KAIST
Daejeon, Republic of Korea
sungmin.kang@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Many works have recently proposed the use of Large Language Model (LLM) based agents for performing 'repository level' tasks, loosely defined as a set of tasks whose scopes are greater than a single file. This has led to speculation that the orchestration of these repository-level tasks could lead to software engineering agents capable of performing almost independently of human intervention. However, of the suite of tasks that would need to be performed by this autonomous software engineering agent, we argue that one important task is missing, which is to fulfil project level dependency by installing other repositories. To investigate the feasibility of this repository level installation task, we introduce a benchmark of of repository installation tasks curated from 40 open source Python projects, which includes a ground truth installation process for each target repository. Further, we propose INSTALLAMATIC, an agent which aims to perform and verify the installation of a given repository by searching for relevant instructions from documentation in the repository. Empirical experiments reveal that that 55% of the studied repositories can be automatically installed by our agent at least one out of ten times. Through further analysis, we identify the common causes for our agent's inability to install a repository, discuss the challenges faced in the design and implementation of such an agent and consider the implications that such an agent could have for developers.**

***Index Terms*—LLMs, installation, documentation**

I. INTRODUCTION

Large Language Models (LLMs) are statistical language models, typically based on the Transformer [1] Deep Neural Network architecture, that are trained on large amounts of data with the goal of predicting the next token in a sequence of text. LLMs trained with large corpora have shown emergent capabilities [2], including in-context learning [3], i.e., the ability to perform tasks for which the models have not been explicitly trained. In particular, LLMs are capable of software related tasks when the training corpora include source code [4].

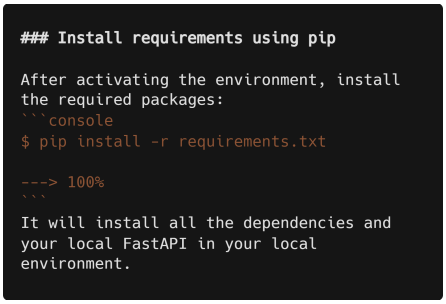
LLMs have been rapidly adopted into software engineering [5]. Initially, the target applications were of small, local scopes: synthesizing individual functions [4], mutating inputs for fuzzing [6], generating a unit test case [7], and generating single-line patches [8], etc. As increasingly larger models are being developed and made available [9], more advanced prompting techniques (such as Chain-of-Thoughts [10], ReAct [11],and self-consistency [12]) and LLM-based architectures (such as agent [13] and multi-agent architectures [14], [15]) have been adopted to perform "repository-level" tasks [16], [17], which we define in this paper to be tasks that require reading and/or writing multiple files in a given repository.

We argue that all of these repository level agents almost exclusively focus on *code management* tasks, i.e., tasks that analyse or manipulate the source code of a repository. On the other hand, developers also frequently deal with environment management, for which LLM based agents have not been applied to, to the best of our knowledge.

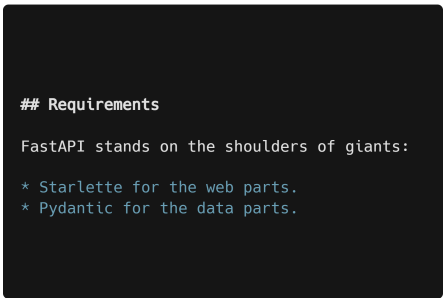
Thus, to fully understand how LLM based agents could aid developers in practice, it is imperative to investigate their ability to perform *environment management* tasks. To this end, this paper presents a novel task for LLM-based agents, which is to install a given code repository, as well as to validate the installation. Despite it being a common task for many developers, attempts to automate the installation of open-source projects have not been made in previous works. Dagenais et al. [**?**] found that inadequate deleloper documentation is an obstacle for new newcomers to a software project. Moreover, in a survey by Aghajani et al. [18], 68% of developers asked said that incomplete documentation of the installation, deployment and release of a project is an important issue, and 63% claimed that inappropriate installation instructions were a common issue as well. As such, we believe that a tool that can be used to attempt automatic installation could lessen developer frustration and improve productivity.

In order to evaluate LLM-based agents for this task, we create a dataset of 40 open source Python repositories to serve as a benchmark for the performance of environment management agents. Each repository is assigned a set of tags indicating its installation method and a ground truth working installation file. To ensure the safe execution of LLM generated installation scripts, we also provide an interface for connecting the LLM agent to a virtual machine in which to safely attempt installation. We make our research artefact, including this dataset, publicly available to facilitate the continued evaluation of LLM based agents: https://github.com/coinse/installamatic.

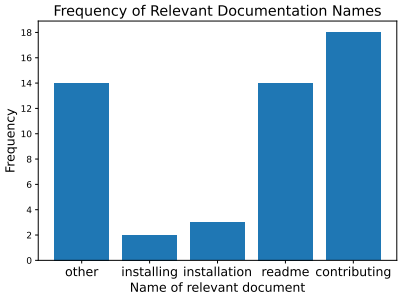
Using this dataset, we investigate how well an LLM-based agent can resolve the installation task by creating INSTALLAMATIC, which attempts to automatically install and test repositories based on their contents. Given a target repository, the agent makes use of search tools to navigate through its contents, and inspect files to find information relevant to the



(a) Example of an install-relevant piece of documentation.



(b) Example of a non install-relevant piece of documentation. (lightly edited for clarity)



(c) Frequencies of install-related document filenames

Fig. 1: Inspection of repository contents

installation or testing process. Once the agent has completed its documentation gathering, we prompt it to write a Dockerfile that, when placed inside the target repository, installs any required dependencies and runs the test suite of the target repository to confirm the success of the installation.

Our empirical evaluation of INSTALLAMATIC's ability to perform the installation task shows that our LLM-based agent can successfully install 21 out of 40 repositories, achieving the success rate of 55%. Through further experimentation, we present lessons for designers of future environment management agents, as well as open source project maintainers. For future development of environment management agents, we suggest the inclusion of a repair step after an agent's initial installation attempt. For maintainers of open source repositories, we suggest providing code examples of the installation process when writing installation instructions in the repository's documentation. We also identify several challenges specific to environment manipulation tasks, such as gathering task-relevant information from the repository.

The technical contributions of this paper are:

- A dataset of 40 open source Python repositories, designed to serve as a benchmark for evaluating the effectiveness of repository-level agents' understanding of repository contents and environment management tasks.
- The initial design of a repository-level agent, INSTALLAMATIC, capable of searching for and reading documentation, then writing a Dockerfile to install and test the repository based on the gathered information.
- An outline of the key challenges that future researchers will likely face when developing a repository-level agent for documentation.

The remainder of the paper is structured as follows. Section II describes the benchmark construction, analysis and labelling process; Section III provides a breakdown of our proposed agent, INSTALLAMATIC; Section IV presents the three research questions we answer, as well as define key metrics in our analysis; Section V examines the results of our experiments; Section VI compares our work with related literature; Section VII discusses the threats to validity; and Section VIII concludes.

II. DATASET CONSTRUCTION

In order to create a benchmark for the task of automatic installation, we collect and present a dataset of 40 open source Python repositories from GitHub and provide the correct method of installation, as well as the location of any relevant documentation, for each repository. In cases where no relevant documentation was found, the appropriate installation method was identified through inspection of non-documentation files in the repository and manual attempts to install the repository.

Repositories were sampled using the GitHub API from several different ranges of stars: 1k-5k, 5k-10k, 10k-20k and >20k. From each range, the 10 most recently updated repositories at the time of collection were chosen, meaning that all repositories in the dataset have been in active development until at least August of 2024. The dataset contains the commit ID that corresponds to the time of collection.

We only choose repositories with test suites located in a `test` or `tests` directory, to ensure that we have a consistent oracle to determine the success of the installation process. We argue that successful executions of test suites can serve as objective and automatable evidence for successful installations. While this decision was necessary to evaluate the capability of our agent, it does bias our sample towards the repositories with test suites, an issue that is discussed further in Section V-C. Table I shows the collected repositories.

Each repository has been manually inspected to produce three different types of metadata: the aforementioned list of all install-relevant documents, an exemplar Dockerfile that successfully installs and tests the repository, and a set of tags indicating the expected installation and testing methods. The Dockerfile writing and tag assignment process is described in more detail in II-B.

A. Documentation Structure of Open Source Python Projects

We define a document to be 'install-relevant' if it makes explicit references to the process of installing the target repository's dependencies and executing its test suite. For example, Figure 1a is an example of install-relevant documentation, as it clearly shows one of the the commands needed to be run to set up a development environment. On the other hand, Figure 1b shows an example of a non install-relevant piece

TABLE I: List of repositories in the dataset

⋆	Name	URL	⋆	Name	URL
1k-5K	icloud-drive-docker	https://github.com/mandarons/icloud-drive-docker	10k-20K	yfinance	https://github.com/ranaroussi/yfinance
	django-stubs	https://github.com/typeddjango/django-stubs		beets	https://github.com/beetbox/beets
	pennylane	https://github.com/PennyLaneAI/pennylane		starlette	https://github.com/encode/starlette
	X-AnyLabeling	https://github.com/CVHub520/X-AnyLabeling		datasets	https://github.com/huggingface/datasets
	opencompass	https://github.com/open-compass/opencompass		mypy	https://github.com/python/mypy
	R2R	https://github.com/SciPhi-AI/R2R		sympy	https://github.com/sympy/sympy
	Torch-Pruning	https://github.com/VainF/Torch-Pruning		ydata-profiling	https://github.com/ydataai/ydata-profiling
	scvi-tools	https://github.com/scverse/scvi-tools		spotify-downloader	https://github.com/spotDL/spotify-downloader
	sabnzbd	https://github.com/sabnzbd/sabnzbd		qlib	https://github.com/microsoft/qlib
	dlt	https://github.com/dlt-hub/dlt		scapy	https://github.com/secdev/scapy
5k-10K	camel	https://github.com/camel-ai/camel	20K+	fastapi	https://github.com/tiangolo/fastapi
	boto3	https://github.com/boto/boto3		black	https://github.com/psf/black
	cloud-custodian	https://github.com/cloud-custodian/cloud-custodian		tqdm	https://github.com/tqdm/tqdm
	aim	https://github.com/aimhubio/aim		rich	https://github.com/Textualize/rich
	speechbrain	https://github.com/speechbrain/speechbrain		open-interpreter	https://github.com/OpenInterpreter/open-interpreter
	nonebot2	https://github.com/nonebot/nonebot2		core	https://github.com/home-assistant/core
	moto	https://github.com/getmoto/moto		sherlock	https://github.com/sherlock-project/sherlock
	instructor	https://github.com/jxnl/instructor		spaCy	https://github.com/explosion/spaCy
	numba	https://github.com/numba/numba		you-get	https://github.com/soimort/you-get
	pymc	https://github.com/pymc-devs/pymc		textual	https://github.com/Textualize/textual

of documentation from the `README.md` file of the FastAPI repository: this is not install-relevant as it does not affect the installation process for a developer.

After identifying the install-relevant documents for each repository, we have found 29 unique file paths leading to install-relevant documentation, and 18 different names for files containing install-relevant documentation, ignoring differences between file type, case sensitivity and the use of '−' and '_'. Figure 1c shows the distribution of documentation file names. While one may expect the 'readme' file of a repository to contain information relevant to the installation of a repository, we find that this is often not the case.

Names 'contributing' and 'readme' are considerably more common than any other file name: files with these names contain install-relevant information in 40% and 35% of repositories in the dataset, respectively. After these two, there are no other file names which are install-relevant and occur more than three times. Note that the purpose of a 'contributing' file is to instruct developers new to the project how to go about making contributions to the project. As such, instructions on dependency management, environment setup and testing are commonplace in files named 'contributing'.

It is worth noting that many projects host much of their documentation on external websites, though occasionally the source files for these external websites are stored in the repository itself. In such cases, the documentation will be often stored inside a `docs` directory, meaning that it is still accessible in the repository. However, it is considerably less visible and consequently harder to find, compared to documentation stored in the repository's root directory. The diversity in possible locations for install-relevant documentation shows that there is no agreed upon structure for documentation of Python repositories.

B. Methods of Installation and Testing

In addition to identifying the locations of install-relevant documentation, our dataset contains a Dockerfile that would, when placed inside of the target repository, install any dependencies, and run the test suite to confirm their successful installation. During this process, we create a series of coarse-grained categories for the different types of commands used during installation. Table II lists the resulting 17 tags.

After assigning appropriate tags to each of the 40 repositories, we are left with 31 unique combinations of tags, meaning that there are 31 different methods of installing the dependencies and running the tests of the Python repositories we sampled. Such a wide variety of installation methods leads us to consider the use of Large Language Models (LLMs) as a means of automatic installation; their in-context learning capability [19] makes them an ideal candidate in tasks like this, where the expected output (in this case, a working Dockerfile) can vary considerably, and is highly dependant on the contents the documentation in the repositories.

III. INSTALLAMATIC: AUTOMATIC INSTALLATION AGENT

This section presents INSTALLAMATIC, an LLM-based agent that attempts to automatically install a given open source Python project. INSTALLAMATIC performs its task in two stages. First, it gathers documentation related to the installation. Second, it tries to build and repair a Dockerfile[4] to install and test the target repository. This section first explains how INSTALLAMATIC searches for files relevant to installation process, and subsequently describes the two stages of its task.

A. Repository Search Process

INSTALLAMATIC needs to search through the contents of the repository during both the documentation gathering and the Dockerfile build/repair step, although, its exact goal can differ in these steps. To this end, we equip INSTALLAMATIC with a generic method to search through the contents of the repository, which can be reused in different stages. An overview of this search method is shown in Figure 2a. Inspired by previous works finding benefits such as improved

[4]https://www.docker.com/

TABLE II: List of Installation Tags

Tag	Description	Tag	Description
requirements	Installation of dependencies using `pip install -r requirements.txt`.	**install-other**	Installation of dependencies through means not listed above, such as a custom script contained in the repository
requirements-extra	Installation of dependencies from additional requirements files, such as `requirements-test.txt`.	**pytest**	Tests are run using pytest.
pip-extra	Requiring the installation of something other than Poetry or the contents of a requirements file.	**pytest-extra**	Additional arguments need to be provided to pytest, such as specifying the location of the tests or additional flags.
Poetry	Installation of dependencies using the Poetry dependency manger.[1]	**tox**	Tests are run using Tox.
Poetry-extra	Installation of dependencies using Poetry, with additional arguments, e.g. `poetry install --no-interaction --with sentry-sdk`.	**unittest**	Tests are run using Python's built in unittest command.
make-install	Installation of dependencies using a makefile, typically commands such as `make install` or `make init`.	**make-test**	Tests are run using a makefile with a command such as `make test`.
install-self	The project itself needs to be installed in the working environment, e.g. by running `pip install -e .`	**test-other**	Tests are run some other way, such as a `test.py` file.
install-pytest	The Pytest[2] library needs to be installed manually.	**bash-extra**	Requiring additional bash commands to set up the repository, such as creating new directories or granting permissions to certain files.
install-tox	The Tox[3] library needs to be installed manually.		

performance [20] and explainability [21] by emulating human behaviour, we chose to use an LLM-guided search step over traditional search methods, such as BM25 [22] or neural embeddings [23].

We include the contents of the target repository's root directory in the prompt at the start of the search process, then provide several tools that enable the agent to further navigate through the contents of the repository. The four basic functions we provide to INSTALLAMATIC for navigation are as follows:

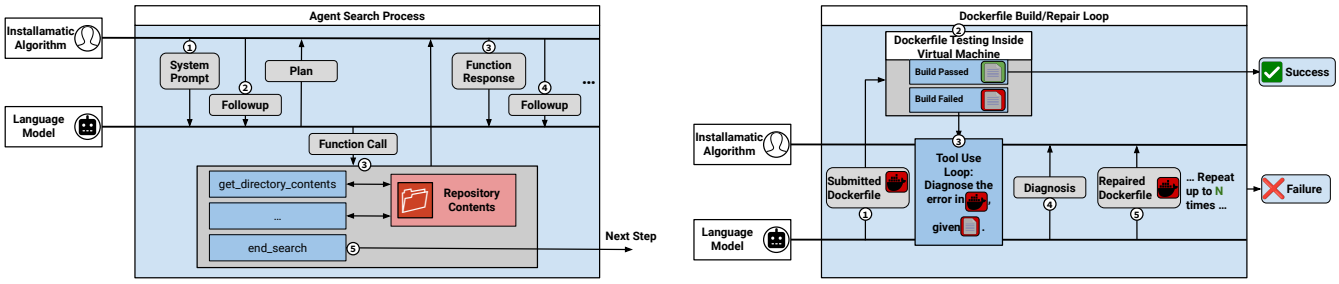
- **get_directory_contents**: Given a path to a directory the agent has already seen, returns the names of all files and sub-directories within that directory.
- **get_file_contents**: Given a path to a file, returns its contents. If the file is human readable and structured, such as a `.md` or a `.rst` file, then the section headers are extracted and returned instead.
- **inspect_header**: Given a path to a file and a name of a section in that file, returns the contents of that section. This is used to minimize the amount of distracting contents shown to the agent.
- **check_presence**: Given a filename, checks whether it exists. This is provided as a sanity check, to prevent the agent from hallucinating files that are not there.

In addition to these functions that are provided during search, INSTALLAMATIC can access additional functions that are available for specific search tasks it is performing. Examples of these specific tasks will be provided below. All prompts used in this process are listed in the appendix, available in the research artifact.

To start the search process, the agent is shown a system prompt explaining its task and providing the contents of the target repository's root directory (Fig. 2a①). The agent then enters a search loop, starting with being sent a query asking the agent to plan its next move in natural language, followed by a second query offering the agent tools with which it will carry out its plan (Fig. 2a②). This two step approach is designed both to improve the reasoning ability of the agent, inspired by previous work on chain of thought reasoning [10], and to provide a better, more understandable explanation of the agent's behaviour when examining the results. After these two steps, a function is chosen based on the agent's response, and executed to return the relevant result to the agent (Fig. 2a③). Once the result of the function has been sent to the agent, the prompt to make the agent plan its next move is sent again (Fig. 2a④). This process is repeated until the agent deems the search to have ended (Fig. 2a⑤). Additionally, the system (Fig. 2a①), follow-up (Fig. 2a②)), and function response prompts (Fig. 2a③), are changed depending on the specific task the agent is currently performing For example, the follow-up prompt during the documentation gathering step will remind the agent to use the provided tool for recording documentation whenever it identifies a document as install-relevant. In contrast, when performing a diagnosis of an error during the dockerfile repair process, the follow-up prompt will instead remind the agent to stop searching through the documentation once it has gathered sufficient information to suggest a fix to the Dockerfile. By adopting this modular approach, we are able to clearly define different states in which the agent is operating, which has been shown to aid the agent's decision making when given a complex task [24].

B. Documentation Gathering Step

During the first stage, INSTALLAMATIC is tasked with searching through the repository and identifying any files that it considers to have information relevant to either the installation or testing process. In addition to the four basic functions, the agent is given more tools: **submit_documentation**, which records a document as being install-relevant, and **finished_search**, which signals the end of this stage and the beginning of the next step.



(a) Diagram of the agent's search process

(b) Agent control during Dockerfile test/repair step

Fig. 2: Diagrams of INSTALLAMATIC's processes

C. Dockerfile Build/Repair Step

After documentation gathering step, we task the agent to summarise the gathered information in natural language. INSTALLAMATIC is once again given access to the basic search functions. In this stage, it can only access the files that it previously selected as being install-relevant. Once it has finished the search process and used the **submit_summary** tool to give its summary, it is prompted to write a Dockerfile to install dependencies and run tests.

Figure 2b illustrates the Dockerfile testing and repair process in detail. After a Dockerfile is generated by INSTALLAMATIC (Fig. 2b①), it needs to be tested in a safe environment. Since an installation process may involve unknown code and may need admin permissions, a degree of risk is involved when arbitrarily running installation scripts. As such, we choose to test the agent's installation process using Dockerfiles running in virtual machines, thus mitigating the side-effects of running potentially harmful or disruptive code (Fig. 2b②).

Once the Dockerfile build process has finished, the resulting logs are analysed. Installation is considered successful if tests are run, and at least one test passes. Such a result is indicative of a successful installation as the tests running at all implies that dependency issues were not encountered during the setup of the test suite. This is an imprecise measure of build success as, while requiring all tests to pass could lead to false negatives as tests could fail due to issues unrelated, such as API keys not being set, cases where different modules of a project have different requirements could incorrectly pass, resulting in false positives. In the case that any tests passed, the process has finished and the agent has completed its task. If no tests passed or the installation process failed at an earlier step, INSTALLAMATIC starts the Dockerfile repair process.

The repair process starts with a new system message including both the Dockerfile the agent previously submitted, as well as the build logs of the failed installation process (Fig. 2b③). The agent is subsequently instructed to explain what the error message means, and to identify the cause of the error.

Similarly to the previous steps, the agent is given access to the basic search functions. However, the agent is no longer encouraged to inspect only documentation files, but any files it considers to be relevant to the issue. The messages from the previous steps are disregarded, meaning this is effectively a new agent instance, with no knowledge of the previous stages. This is done to reduce the context length of the messages being sent to the LLM, which can both improve its performance [25] and also reduce inference cost.

After providing the explanation why the previous Dockerfile failed (Fig. 2b④), the agent is instructed to suggest how this error could be fixed, as well as the repaired Dockerfile (Fig. 2b⑤), which is then sent to the local virtual machine to be tested again. This process is repeated until INSTALLAMATIC produces a working Dockerfile, or it reaches a fixed number of repair attempts, at which point we consider the attempt to install the repository a failure. Due to the high time cost of additional repair attempts, we set the maximum number of repair attempts to two.

IV. EXPERIMENTAL SETUP

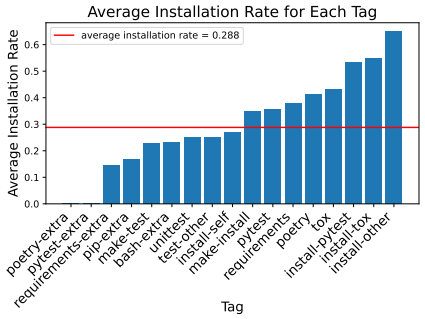
A. Research Questions

In the creation and evaluation of this agent, the following three research questions are posed:

- **RQ1: To what extent is our agent able to install arbitrary Python repositories?**
  This question analyses our agent's performance on our benchmark dataset and aims to determine the circumstances under which our agent is and is not able to automatically install a repository.
- **RQ2: What factors affect an LLM agent's ability to successfully install a repository?**
  Understanding what factors could affect an agent's compatibility with a repository could prove invaluable in the case that the use of LLM agents for repository-level tasks become widespread. As such, we aim to identify the key characteristics of repositories that are easier and harder to automatically install.
- **RQ3: What challenges remain in designing agents for automatic installation and similar repository level tasks?**
  To aid future research in the pursuit of using LLM agents for environment management tasks, we outline the greatest challenges and limitations we encountered during the ideation and development of our agent.

B. Metrics

We aim to study the relationship between the quality of documentations, and the performance of our agent. To estimate



(a) Average install rate for each tag

```dockerfile
FROM python:3.8

WORKDIR /app

COPY . /app/

RUN pip install poetry

RUN poetry install --all-extras

RUN poetry run pytest
```

(b) Exemplar DockerFile for the NoneBot2 repository

```dockerfile
FROM python:3.8

COPY . /app/

WORKDIR /app

RUN pip install -e ".[dev]"
RUN pip install cachetools
WORKDIR /app/tests

RUN python -m pytest . -m "not slow"
```

(c) Exemplar DockerFile for the Qlib repository

Fig. 3: Identifying causes of un-installable repositories

the quality of documentations, we propose two measures of document quality: visibility and informativity.

The visibility of a repository's documentation intuitively measures how easy it is to find the install-relevant documents. For this, we count the total number of files and directories that need to be traversed to reach all install-relevant documents, and take its inverse:

$$\text{visibility} = \frac{1}{\#\text{directories} + \#\text{files}}$$

The informativity of a repository's documentation is, intuitively, a measure of how comprehensive the documentation is with respect to the actual installation process as defined by our ground truth Dockerfile. For this, we compute the proportion of lines of the ground truth Dockerfile that also appear in install-relevant documentations:

$$\text{informativity} = \frac{|\text{dockerfile} \bigcap \text{documentation}|}{|\text{dockerfile}|}$$

Note that these two heuristic measures use the gathered ground truths; thus, their computation relies on manual inspection of the studied repositories. We use these measures in Section V-A to determine the relationship between the agent's success rate and the quality of the documentation, with the expectation that repositories with poorer quality, as estimated by these metrics, will be more difficult for the agent to install and consequently will have a lower average installation rate.

To measure installation performance, we use two other metrics: the recall from its documentation gathering step, and the successful installation rate of the Dockerfile generation step, across repeated runs. The recall is computed as follows:

$$\text{recall} = \frac{|\text{install-relevant retrieved documents}|}{|\text{install-relevant documents}|}$$

Whereas the successful installation rate is simply the proportion of successful installations among the repeated runs. We apply INSTALLAMATIC to each repository 10 times.

C. Implementation

All experiments have been conducted using the GPT4o-mini (`gpt-4o-mini-2024-07-18`) model; the Dockerfiles have been built in a virtual machine running Ubuntu 22.04.4 LTS and Docker 27.1.2. All Python scripts in the artiact were run on Python 3.10.12 and figures' coefficients are calculated using the `polyfit` method of NumPy 2.0.1

V. RESULTS

Here we present the findings from our empirical evaluation of INSTALLAMATIC.

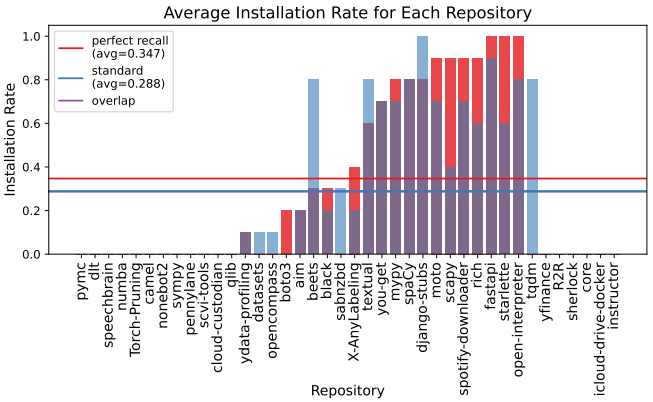
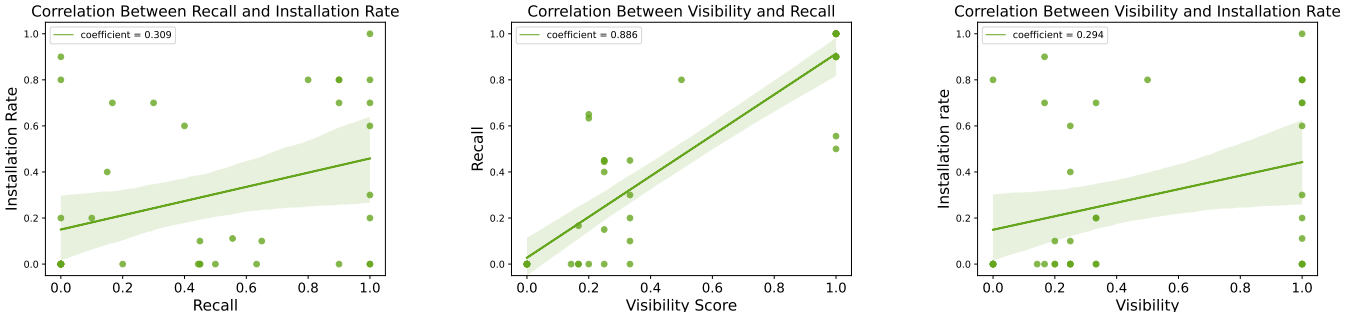


Fig. 4: Successful install rate for each repository, with and without perfect recall of relevant documents (purple bars represent the overlap between these two metrics).

A. RQ1: To what extent is our agent able to install arbitrary Python repositories?

1) Installation Success Rate: Figure 4 shows the rates of successful installations achieved by INSTALLAMATIC for the studied repositories under two configurations. The first configuration, *standard* (blue bars in Figure 4), is the standard configuration of INSTALLAMATIC including the documentation gathering step: the agent searches through the repository and select any items of documentation it deems to be install-relevant before attempting to write a Dockerfile. The second configuration, *perfect recall* (red bars in Figure 4), is INSTALLAMATIC without the documentation gathering step: instead, we provide INSTALLAMATIC with all documents that we manually confirmed to be install-relevant. The second configuration is included to isolate the installation success rate from the visibility of the repository's documentation. The seven rightmost repositories do not contain any relevant



(a) Correlation between the documentation search recall and the average install rate

(b) Correlation between the recall of the documentation search step and visibility

(c) Correlation between visibility and and average installation rate

Fig. 5: Evaluating the visibility of a repository's documentation and its average installation rate

documentation in their repositories and consequently are excluded from the perfect recall configuration. Of these seven repositories, only `tqdm` was successfully built by the agent.

Under standard configuration, INSTALLAMATIC can successfully install 21 of 40 tested repositories at least once, with a 28.8% average successful installation rate across all repositories. Under the perfect recall configuration, INSTALLAMATIC can install 18 of 34 tested repositories, with a 34.7% average installation rate, a roughly 20% increase. Of the 34 repositories tested in both configurations, 13 are never successfully built.

Note that there are several repositories whose installation rate is higher in the standard configuration than in perfect recall configuration. This is likely due to variance in the LLMs beahaviour (as only 10 attempts were made per repository), rather than the repository actually being easier to install with imperfect recall of install-relevant documentation.

2) Causes of Failure: In order to identify the limitations of the agent's capabilities, we consider the relationship between each of the 17 previously defined tags with the installation success rate of the repository in Figure 3a. Of the tags present across all 40 repositories, there are only two which the agent is never successful in installing: **poetry-extra** and **pytest-extra**; example installation processes that include these tags can be seen in Figures 3b and 3c respectively.

Figure 3b shows the exemplar Dockerfile written for to install and test the NoneBot2 repository. In this example, the additional `--all-extras` argument needs to be added to the installation command as the repository's test suite tests modules with dependencies not installed by `poetry install` on its own. Despite this, the `--all-extras` tag is not mentioned in the install-relevant documentation, resulting in the agent not being aware of this problem, even in the case when the documentation is provided.

Figure 3c shows a Dockerfile for the `Qlib` repository, and features the other un-installable tag, **pytest-extra**. Here, two additions are made to the standard testing process. First, the user must run the test suite from within the `test` directory, and second, the test suite must be run with the `"not slow"` argument. Neither of these additions were mentioned of the documentation for the repository, although, this is to be expected of the second requirement. While first of these two additions is necessary for the test suite to run correctly, the second addition is due to our testing process, rather than the nature of the repository itself. In order to prevent Dockerfile build processes from stalling and never completing, usually due to connection issues, we enforce a 30 minute time limit on the build process; any Dockerfile that takes over 30 minutes to build is interrupted and considered insufficient. As such, it is common for repositories with larger test suites to be run with an additional command such as the aforementioned `"not slow"` to allow for the testing process to finish in time. While important to guarantee that our experiments eventually finish, the requirement for test suites to finish within 30 minutes nonetheless is a detriment to our agent's installation rate, as without it, it could be the case that repositories such as `Qlib` could be successfully built.

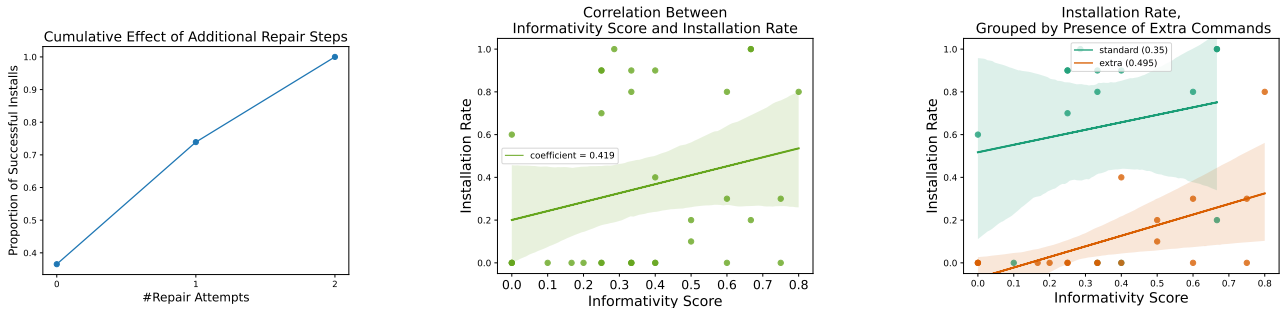
In both of these cases, the lines or additional arguments that are required for the installation process to succeed are not mentioned in the install-relevant documentation. While these two tags are not present in all un-installable repositories, we were able to confirm that the presence of these two tags were the most common reason for failure of these repositories.

Answer to RQ1: INSTALLAMATIC can successfully install 21 out of 40 studied repositories, with the average installation success rate of 28.8%. Most common reason for failed installation is that a successful installation process requires commands or arguments that are not mentioned in the relevant documentation.

B. RQ2: What factors affect an agent's ability to successfully install a repository?

The first factor we consider is that of INSTALLAMATIC's own repair step. Figure 6a shows the proportion of successful installations that occur within each repair step. Only 36.5% of successful installations (10.5% of all installation attempts) occurred without the need for any repairs, increasing to 73.9% after a single repair attempt (21.3% of all attempts). These results indicate that further repair steps could potentially increase the installation rate, though this would come at the cost of increased installation time, as will be discussed in RQ3.

Figure 5a shows the relationship between the average recall of install-relevant documents and the installation rate of each



(a) Proportion of successful installations for each repair step (cumulative)

(b) Correlation between the informativity score and the installation success rate

(c) Effect of extra commands on the installation rate of a repository

Fig. 6: Factors affecting a repository's installation rate

repository. The correlation is positive, confirming our intuition that the documents we marked as install-relevant do indeed assist the agent in successfully installing the repository. Figure 5b shows a strong positive correlation between the average recall and the visibility of a repository. Consequently, the correlation between visibility and installation rate is similarly positive, shown in Figure 5c. This consistency between the relationships of recall and visibility with the agent's installation rate leads us to conclude that the structure of a repository's documentation does have an effect on the ability of our agent to install the repository; documentation made of fewer files, stored within fewer subdirectories, will be more compatible with LLM-based agents.

The second of our two proposed document quality metrics, informativity, also correlates with our agent's ability to install a repository. Figure 6b shows the correlation between the informativity score of a repository – the number of lines in our exemplar Dockerfile that appear in the repository's install-relevant documentation – and the average installation rate. In order to isolate the effect of informativity on the installation rate, the results shown here are from the experiment in which all install-relevant documents are given to the agent in place of the documentation gathering step. Similarly to visibility, the correlation between informativity and installation rate is positive, indicating that the agent responds well to documentation that contains lines of code that appear in the necessary installation steps. However, informativity does not consider other features of documentation, such as natural language instructions. Consequently, it is unclear whether instructions other than lines of code are more or less effective in supporting the agent's installation attempts.

Comparing the agent's installation rate only to our metrics of document quality - recall, visibility, and informativity- would neglect the effect that the complexity of a repository's installation process has on the performance of the agent. Unfortunately, the complexity of an installation process is hard to quantify, and so too is the process of identifying and mitigating the effect of difficult to install repositories when evaluating our agent. Instead, here we aim to do a qualitative estimation of the difficulty of a repository's installation process based on the types of tags it has been assigned with. Figure 6c shows a clear difference in build rate of repositories that are tagged with **extra** (**pip-extra**, **requirements-extra**, **poetry-extra**, **pytest-extra** or **bash-extra**) and those that are not. It can therefore be inferred that, while the agent may be familiar enough with tools such as `pip`[5], Poetry or PyTest to perform basic tasks, the inclusion of additional complexity, such as the need of specific additional flags or installing requirements from multiple files greatly reduces the agent's ability to write a correct Dockerfile to install the repository. We note that this increased difficulty can be curbed to some extent through the addition of thorough documentation on the installation process, as demonstrated by the positive correlation between informativity and the installation rate observed from those repositories tagged with **extra**.

Answer to RQ2: The inclusion of additional complexity to a repository's installation process has the greatest impact on the installation rate, though this effect can be mitigated through clear documentation structure and the use of code examples in the installation instructions.

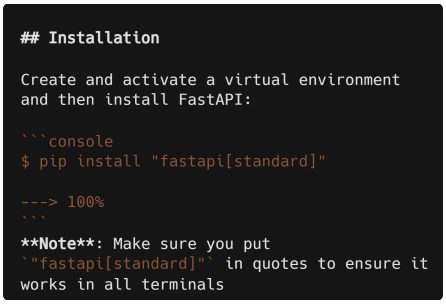
C. RQ3: What challenges remain in designing agents for automatic installation and similar repository level tasks?

Through the process of creating and evaluating this agent, we faced four main challenges which limit both the effectiveness and practical feasibility of our agent.

1) Identifying install-relevant documentation: As reported in Section II-A, install-relevant documentation can be found in various locations within a repository. INSTALLAMATIC shows recall of zero for 11 out of 40 repositories, despite there being only six repositories in the dataset without any install-relevant documentation. The lack of consistent structure in documentations of open source Python repositories affects the ability of LLM-based agents to identify install-relevant documents. RQ1 and RQ2 show that the failure to retrieve install-relevant documents negatively affects the installation rate of the repository substantially.

An example of how irrelevant information can lead to incorrect Dockerfile can be found in the 'Installation' section of the the `README.md` file of `FastAPI`, shown in Figure 7a. The section provides installation instructions for

[5]https://pypi.org/



```
## Installation

Create and activate a virtual environment
and then install FastAPI:

```console
$ pip install "fastapi[standard]"

---> 100%
```
Note: Make sure you put
`"fastapi[standard]"` in quotes to ensure it
works in all terminals
```

(a) Example of documentation that is not relevant to setup of a development environment

```
...
# Install dependencies from requirement file
RUN pip install "fastapi[standard]"
RUN pip install pydantic starlette

# Run the tests to verify everything works
RUN pytest
```

(b) Incorrect Dockerfile generated for FastAPI

```
...
# Run the tests to confirm everything is
working
RUN poe test


# Corrected to

...
# Run the tests to confirm everything is
working
RUN poetry run pytest
```

(c) Incorrect Dockerfile generated for beets

Fig. 7: Examples of distracting documentation and incorrect Dockerfiles

users intending to use `FastAPI` to develop their own tools, rather than those who want to work on the `FastAPI` code itself. While it may be clear to a human developer whether this section is relevant to them, our evaluation of INSTALLAMATIC shows that LLM-based agents may be susceptible to misunderstanding the intention of this document, which is shown in Figure 7b, where the generated Dockerfile erroneously follows the instructions of the irrelevant section, rather than correctly installing the dependencies of the project from the `requirements.txt` file. In this case, the installation failed when trying to run PyTest, as Pytest is not included in `"fastapi[standard]"`, `pydantic` or `starlette`.

2) Writing valid Dockerfiles: INSTALLAMATIC often fails to write a correct Dockerfile, not because it misunderstands the installation instructions, but simply because it made mistakes while writing the Dockerfile itself. Figure 7c gives an example, i.e., a typo made by INSTALLAMATIC in one of the generated Dockerfiles for the `beets` repository. As shown, this typo was identified and fixed by the agent during the repair step.

While we maintain that attempting to install the repositories inside a Docker container is necessary to mitigate the effect of any potentially harmful code, installation and testing in the manner we instruct the agent to do is not the usual purpose of Docker, and we suspect that this unfamiliarity with writing installation scripts in a Dockerfile could lead to additional mistakes. A previous work [26] has encountered a similar issue of an LLM agent being prone to low-level mistakes, despite seeming to have a high level understanding of the given task.

3) Cost: In previous works, repository level techniques that search through the contents of a repository delegate this search process to static analysis tools [17] or a non-LLM based search technique, such as BM25 [15], and then include the search results in a prompt to the LLM. In order to emulate the behaviour of a human developer, our search process is controlled by the LLM itself. This means that the number of tokens used by the agent is largely unconstrained, resulting in considerably higher usage than related work. The additional cost can vary greatly depending on the contents of the target repository; extensive documentations or ambiguous naming can lead to the agent searching through more files than necessary, and thus consuming more tokens [25], [27], [28], resulting in higher cost. Aside from the financial cost incurred, the time taken by the agent to attempt an installation of a repository can vary considerably depending on the size of its dependencies, its test suite and of the repository itself; an installation attempt takes 501 seconds on average with INSTALLAMATIC, with the longest run taking almost 80 minutes over the course of Dockerfile building attempts.

4) Oracle problem: While we choose to use a repository's test suite for the oracle, this is not an ideal solution for various reasons. It is not a trivial task for any agent to run tests for an arbitrary project, as Python supports several different methods of testing. Any agent aiming to run tests for an arbitrary Python project should first correctly identify the method of test execution, on top of correct identification of the installation method, adding an extra layer of complexity to the overall task. Additionally, it is also possible that a repository does not contain any test, making it impossible for our current design of INSTALLAMATIC to attempt installation.

Figure 3a shows that all testing related tags other than **pytest** contribute to a lower installation rate. In the case that these alternative testing methods are the cause of this reduced installation rate, using an oracle other than running the test suite would allow us to ignore the effect of these tags, as the agent's performance would no longer be dependant on its ability to run tests.

Answer to RQ3: There are several limitations faced when using an LLM based agent to automatically install Python repositories, with the most critical being finding an oracle that is both generalisable and accurate.

VI. RELATED WORK

A. Repository-Level Tasks

Bairi et al. [16] define an LLM-driven repository-level coding task as one that requires a series of edits to be made to the state of a code base until some oracle is satisfied. Previous works have proposed the use of LLMs to perform tasks that satisfy this definition of repository-level coding tasks [15], [29], and are typically evaluated on benchmarks such as SWE-bench [30]. SWE-bench tasks a language model to edit a codebase to address a description of an issue. These edits are then assessed using a test suite, as well as through comparison with a human written pull request that resolves the issue.

This task of issue resolution posed in SWE-bench is a clear example of a repository-level coding task as described by Bairi et al. [16]. However, the definitionof Bairi et al. is rather narrow, and does not consider other examples of repository-level software engineering that has been proposed in recent works. AutoFL [13] is a fault localization technique that makes use of function calls to search through the repository and find potentially erroneous lines. It fits our definition of repository-level task, as it makes use of information across mlutiple files in a repository, but not that of Bairi et al., as AutoFL is a debugging technique and does not make any edits. Other works propose techniques that consider multiple files in a repository, but are only tasked with generating code for a single function [17]. In tasks such as this, the scope of the changes is limited to a single file, but the agent is nonetheless able to access files across the whole repository.

B. Automated installation

To the best of our knowledge, no previous literature has addressed the challenge of automatic installation of arbitrary repositories, although there are some works that have addressed similar tasks. A recent work by Guerrero, Corcho and Garjio [31] proposes PlanStep, a methodology to extract structured installation instructions from README files of research software projects through the use of LLMs. While this project is very similar to the initial search step of our proposed agent, PlanStep's goal is to identify all possible methods of installation, rather than performing the installation itself.

Cognition AI's Devin[6] project, which claims to be an 'AI software engineer', does seem capable of automatically installing an open-source repository; Devin has been demonstrated cloning a repository and installing its dependencies, given only the GitHub URL of the repository. Unfortunately Devin is not publicly available at the time of writing, so its exact capabilities are currently unclear.

C. Documentation analysis with LLMs

In a recent survey [32] on the use of LLMs for software engineering tasks, while hundreds of papers on LLMs for software engineering were identified, none used documentation as input to perform some task, and only one addressed the task of evaluating the quality of a code-base's documentation. Furthermore, the work in question [33] by Khan et al. specifically focus on detecting API Documentation smells, rather than high level documentation in a repository, and as such is not relevant to the technique we propose.

Liang et al. [26] investigates the use of another form of documentation, empirical software engineering papers, with the goal of replicating their research methodologies and results. A key finding of this paper was that, when writing code to replicate the contents of these papers, the LLM they used (GPT-4) "is correct in its high-level structure, but can contain errors in its lower-level implementation". While neither the input nor the output of this task is directly comparable to the automatic installation of python repositories, we experienced a similar problem with the LLM that we use, GPT-4o-mini, as we discussed in Section V-C2.

[6]https://www.cognition.ai/blog/introducing-devin

VII. Threats to Validity

Threats to internal validity are challenges to the findings of the paper. Due to the stochastic nature of LLMs, the behaviour and thus performance of our agent can vary between runs. To mitigate this randomness we repeat our experiments 10 times, and report the average scores over these 10 runs. For reproducibility, we make both our implementation and the messages generated in our experiments available for scrutiny.

Threats to extrenal validity concern whether the reported findings may generalise to other results. The design of our agent allows for it to be applied to repositories not contained on our dataset, although the calculation of metrics such as recall would require additional manual inspection. We also design our agent to be agnostic to the LLM used to control it, allowing for experimentation with different models.

Threats to construct validity concerns whether the measurements are actually based on the properties we are interested in. The quality of installation documentation, as well as their usability in terms of repository organization, is a highly abstract property and can be subjective. We aim to define clear and transparent heuristic measures for these with visibility and informativity. Other metrics such as recall and installation rate are both intuitive and straightforward.

VIII. Conclusion

This paper studies a novel repository level task for LLM-based agents, namely automated installation of arbitrary repositories. In order to provide insight into this task, we make three contributions. First, we created a dataset of 40 open source Python repositories for evaluating the effectiveness of repository level agents' understanding of documentation, as well as their ability to correctly install a repository. Second, we present InstallAMatic, an easily adaptable design for an LLM-based agent that is able to autonomously inspect the contents of a repository and recover items of documentation relevant to its task. An empirical evaluation of InstallAMatic using our dataset shows that it can install 21 out of the 40 repositories, as well as a clear correlation between the structure and content of a repository's documentation and the performance of our agent. Finally, we report the challenges faced when developing an agent for repository-level agents for documentation,as well as suggestions to overcome these challenges for future developers. We hope that our dataset and empirical insights can contribute to future work on environment management tasks such as automated installation.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (RS-2023-00208998), and the Engineering Research Center Program funded by the Korean Government MSIT (RS-2021-NR060080).

REFERENCES

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017.

[2] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph *et al.*, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022, survey Certification. [Online]. Available: https://openreview.net/forum?id=yzkSU5zdwD

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[5] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta *et al.*, "Large language models for software engineering: Survey and open problems," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*, ser. ICSE-FoSE, May 2023, pp. 31–53.

[6] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639121

[7] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[8] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[9] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar *et al.*, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.

[10] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran *et al.*, "React: Synergizing reasoning and acting in language models," in *Proceedings of the International Conference on Learning Representation*, ser. ICLR 2022, 2022.

[12] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi *et al.*, "Self-consistency improves chain of thought reasoning in language models," *CoRR*, vol. abs/2203.11171, 2023.

[13] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1424–1446, 2024.

[14] J. Yoon, R. Feldt, and S. Yoo, "Intent-driven mobile gui testing with autonomous large language model agents," in *Proceedings of the 16th IEEE International Conference on Software Testing, Verification and Validation*, ser. ICST 2024, 2024.

[15] W. Tao, Y. Zhou, W. Zhang, and Y. Cheng, "Magis: Llm-based multi-agent framework for github issue resolution," *arXiv preprint arXiv:2403.17927*, 2024.

[16] R. Bairi, A. Sonwane, A. Kanade, A. Iyer, S. Parthasarathy *et al.*, "Codeplan: Repository-level coding using llms and planning," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 675–698, 2024.

[17] C. Wang, J. Zhang, Y. Feng, T. Li, W. Sun *et al.*, "Teaching code llms to use autocompletion tools in repository-level code generation," *arXiv preprint arXiv:2401.06391*, 2024.

[18] E. Aghajani, C. Nagy, M. Linares-Vásquez, L. Moreno, G. Bavota *et al.*, "Software documentation: the practitioners' perspective," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 590–601.

[19] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan *et al.*, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[20] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan *et al.*, "Reflexion: Language agents with verbal reinforcement learning," 2023. [Online]. Available: https://arxiv.org/abs/2303.11366

[21] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, "Explainable automated debugging via large language model-driven scientific debugging," 2023. [Online]. Available: https://arxiv.org/abs/2304.02195

[22] G. Amati, *BM25*. Boston, MA: Springer US, 2009, pp. 257–260. [Online]. Available: https://doi.org/10.1007/978-0-387-39940-9_921

[23] B. Mitra and N. Craswell, "Neural models for information retrieval," *arXiv preprint arXiv:1705.01509*, 2017.

[24] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," 2024. [Online]. Available: https://arxiv.org/abs/2403.17134

[25] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua *et al.*, "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, pp. 157–173, 2024.

[26] J. T. Liang, C. Badea, C. Bird, R. DeLine, D. Ford *et al.*, "Can gpt-4 replicate empirical software engineering research?" 2024. [Online]. Available: https://arxiv.org/abs/2310.01727

[27] T. Li, G. Zhang, Q. D. Do, X. Yue, and W. Chen, "Long-context llms struggle with long in-context learning," *arXiv preprint arXiv:2404.02060*, 2024.

[28] Y. Zhou, X. Geng, T. Shen, C. Tao, G. Long *et al.*, "Thread of thought unraveling chaotic contexts," *arXiv preprint arXiv:2311.08734*, 2023.

[29] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang *et al.*, "Opendevin: An open platform for ai software developers as generalist agents," *arXiv preprint arXiv:2407.16741*, 2024.

[30] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei *et al.*, "Swe-bench: Can language models resolve real-world github issues?" *arXiv preprint arXiv:2310.06770*, 2023.

[31] C. Utrilla Guerrero, O. Corcho, and D. Garijo, "Automated extraction of research software installation instructions from readme files: An initial analysis," in *International Workshop on Natural Scientific Language Processing and Research Knowledge Graphs*, 2024, pp. 114–133.

[32] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang *et al.*, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[33] J. Y. Khan, M. T. I. Khondaker, G. Uddin, and A. Iqbal, "Automatic detection of five api documentation smells: Practitioners' perspectives," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 318–329.