An exploratory study on architectural smell refactoring using Large Languages Models

1st Gabriele Pandini*, 2nd Antonio Martini†, 3rd Adela Nedisan Videsjorden‡, and 4th Francesca Arcelli Fontana*
*Dept. of Computer Science, University of Milano Bicocca, Milano, Italy
Email: g.pandini@campus.unimib.it, francesca.arcelli@unimib.it
†Dept. of Informatics, University of Oslo, Oslo, Norway
Email: antonima@ifi.uio.no
‡Sustainable Communication Technologies, SINTEF Digital, Oslo, Norway
Email: adela.videsjorden@sintef.no

***Abstract*—Architectural smells are abundant in codebases and regularly hinder the development of stable and maintainable code. Understanding and removing these elements can consume a huge amount of developers' time, who often need to prioritize implementing new features. This causes a substantial increase in Technical Debt, compromising the scalability and maintainability of the codebases, at time bringing the development to a standstill. Meanwhile, the use of Large Language Models for small error correction is constantly growing, bringing the attention of an ever-wider audience to these technologies. This study explores a first approach to use Large Language Models to suggest refactoring for architectural smells, with a focus on Cyclic Dependencies smells. We study the use of detailed prompt and Retrieval-Augmented Generation (RAG) to enhance LLMs, and we study local vs cloud LLMs. The results are promising, also validated with a series of interviews with students and developers, and highlight how additional and precise context is key to enhance the use of LLMs to propose refactoring suggestions. A multi-agent approach seems to be more suited when increasing the complexity of the smells.**

***Index Terms*—Architectural Smell, Refactoring, LLM, RAG**

I. Introduction

As systems evolve, avoiding architectural erosion and debt is difficult. The causes can be different as also the symptoms. One of the symptoms is represented by the presence of architectural smells [28], which have to be detected and managed to avoid a progressive architectural drift. Several tools have been proposed for the detection of architectural smells [10], however no automatic tool is currently available for their removal, to our knowledge. The refactoring of architectural smells represents a challenging problem that can be manually resolved by applying different refactoring techniques. While removing code smells could be easy, refactoring architectural smells can be a complex and time-consuming task.

With the advent of Generative Artificial Intelligence (GenAI), a number of tasks have been automated or made more efficient by receiving suggestions using Large Language Models (LLMs). For example, Google claims that 25% of their code is being automatically generated by AI [32]. Tools integrated within IDE (e.g. Microsoft Copilot) can be used to refactor code smells [13]. Despite these promising results, LLMs have not been explored to help tackle problems that require a large amount of context, for example when refactoring needs to be performed at architectural level.

In this paper, we explore to what extent LLMs can provide support to refactoring a specific type of architectural smells, Cyclic Dependencies (CD). For the detection of smells, we used the Arcan [3] tool, which automatically detects Architectural Smells (AS), including CDs, and provides general strategies for addressing CD smells of different sizes. We leverage this input to generate refactoring suggestions using LLMs and we qualitatively assess their performance with 12 interviewees, 10 master thesis students in the area of software engineers and 2 experienced developers of an italian company. We also test different LLMs (local or cloud) and two techniques related to improving the output of LLMs: more detailed prompts and the usage of Retrieval Augmented Generation (RAG), which provides the LLMs with additional context. Our contributions are threefold:

- An automatic, LLM-based solution to support developers and practitioners in refactoring activities, tailored to address architectural debt caused by CDs.
- An exploration of prompting techniques to increase the LLM performance in solving the task above through various prompt engineering techniques and RAG, followed by a report on their efficiency for low and higher complexity CDs, considering both public and cloud-based models.
- An empirical evaluation of the approach supported by interviews conducted with participants with diverse backgrounds and CDs extracted from both open-source and company-backed projects.

The paper is organized in the following sections: Section II briefly describes related works. In Section III, we describe the study design, including the research questions, and the different phases that we follow in our study. Section IV outlines the obtained results. Section V delves into the limitations of the approach and addresses threats to validity. Finally, we discuss our results and conclude in Section VI.

II. Related Works

In the literature, several works have been proposed on the use of Large Language Models for code refactoring in different contexts.

Shirafuji et al. [2] examine how code can be simplified to improve maintainability and security. To this end, the authors discuss the feasibility of using the Large Language Model (LLM) GPT-3.5 to help students program in less complex Python scripts better, encouraging better user coding habits.

Guo et al. [1] specify how the nature of code review is time-consuming and error-prone, which is a challenge, but its importance cannot be overemphasized. The study examines ChatGPT's ability to refine code based on reviews using the CodeReview dataset. ChatGPT is then compared with the ability of the CodeReviewer tool, confirming that it is more capable of code refinement tasks.

Zhang et al. [13] explores the presence of code smells in Copilot-generated Python code and then evaluates the effectiveness of Copilot Chat in fixing such code smells using different prompts. From this test it was found that a more informative prompt allowed a much higher resolution rate for all eight types of code smell analyzed.

Jimenez et al. [14] argue that language models need better scrutiny as they become more powerful in their applications. They introduce a new evaluation framework, called SWE-bench consisting of 2294 software engineering problems which are derived from 12 major Python repositories' GitHub issues and pull requests.The study shows that current models, including flagship ones such as Claude 2, solve a tiny fraction of these problems (max 1.96%), highlighting how complex this testbed is and how much improvement is still needed to make LLMs more practical and autonomous.

In summary, although studies have shown how LLMs can be used to automate or suggest solutions for code-related tasks, no study has yet explored the use of LLMs to suggest refactoring for AS such as CDs.

III. Study Design

The following section presents a detailed explanation of the conducted study. We present the methodology, techniques, tools and frameworks employed. Our study focuses on the evaluation of the LLMs' ability to suggest CD smell refactorings. We aim to answer the following Research Question:

Overall RQ. To what extent does the use of LLMs help refactor a CD architectural smell? This research question is concerned with understanding whether an LLM is able to propose a refactoring that is clear, reliable, and sufficiently useful to help the developer produce an optimal solution. In particular, we also focus on different techniques related to LLMs (prompt, LLM deployment, RAG) and we therefore have the following additional RQs:

RQ1. To what extent does providing the LLMs with more detailed prompts give better or worse suggestions to refactor a CD architectural smell? This question aims to determine the LLM ability to understand and suggest corrections under different levels of prompt detail. By answering this research question, we find whether the LLM is able to identify and correct the CD starting from the individual files, or if it requires an enhanced LLM prompt with more context.

RQ2. To what extent does using local LLMs give better or worse suggestions to refactor a CD architectural smell than using a cloud-based model? As the development effort for freely available local models increases, this raises the question of whether the use of paid cloud-based models is necessary to achieve quality refactoring.

RQ3. To what extent does the use of Retrieval Augmented Generation (RAG) improve the suggestions given by LLMs on how to refactor a CD Architectural Smell? This question asks about the usefulness of using the RAG technique which allows one to expand the knowledge domain of an LLM without actually retraining it.

A. Overall Study Design

Our study started by interacting with professionals, collecting their requirements, which allowed us to understand if the approach would produce promising results. After, we used a comparative approach, where several experiments were carried out by varying a number of variables (detailed prompt, LLM type, RAG) to observe their effect. In the experiments, we focused our attention on CD smells. CDs occur when two or more architectural components are involved in a chain of relationships. We analyzed CDs smells for three main reasons:

1) **High number of instances**: CD has been recognized as the smell with the greatest presence within projects [7].
2) **Extensive knowledge of the problem**: CD can significantly impact the efficiency of a codebase, its maintainability and testability ([8] [3] [10] [11] [9]).
3) **Variable size**: CD occurs when 2 or more classes are mutually dependent. This allowed us to initially tackle smaller sized CDs and then scale up and modify the resolution technique.

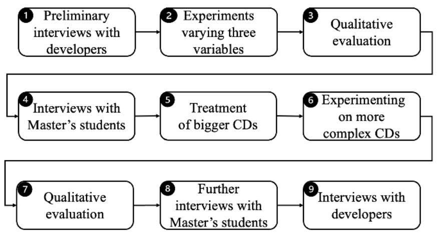
The size of a CD corresponds to the number of files that are affected by the smell. The initial analysis focused on CDs of size 2 as these CDs are easier to understand and manage: breaking a single edge in the dependency graph leads to the resolution of the smell. Furthermore, we focused on investigating CDs refactoring that included concrete classes and enumerations, since breaking inefficient connections results in more robust code. Subsequently, CDs of size 3 and 4 characterized by the *Circle* shape [3] were considered.

For the evaluation of the comparative approaches, we first used the researchers' own expertise, then refined by interviews with software engineering Master students and developers of a company who judged the practical usefulness of the approach. In the following, we report the tools and libraries used during our study, and explain each phase of the study design, as outlined in Fig. 1.

B. Tools and Libraries

a) ***Arcan:*** [1] Arcan is an automatic tool for technical debt computation based on the detection of architectural smells [3]. The Arcan(v2.9.4) version identifies four types of AS: Cyclic Dependency (CD), Unstable Dependency (UD), God

[1]Arcan



1 Preliminary interviews with developers
2 Experiments varying three variables
3 Qualitative evaluation
4 Interviews with Master's students
5 Treatment of bigger CDs
6 Experimenting on more complex CDs
7 Qualitative evaluation
8 Further interviews with Master's students
9 Interviews with developers

Fig. 1. Organization of the study phases

Component (GC), and Hublike Dependency (HL). Each AS identified by Arcan is characterized by a Severity value, which estimates how critical the respective smell is for the project. The Severity is based on the characteristics of the smell (the size and shape) and is calculated using a proprietary machine learning model that has been trained on manually classified examples [28]. CDs can be located on units and containers, such as in Java Classes and Packages, respectively. A key feature of Arcan is the possibility to visualize the AS through dependency graphs, providing the developer with a clear overview of the codebase and at the same time of the specific files that need more attention. The tool has been used in different studies (such as [28] [5] [6] [15]).

b) ***LangChain:*** [2] Langchain is an open-source library that allows the development of projects that make use of Large Language Model (LLM) in a modular and scalable way. It allows customization of the use of a LLM according to a series of characteristics, namely Chains, Prompts, Vector Stores, Retrieval, and Retrieval-Augmented Generation (RAG).

c) ***Ollama:*** [3] Ollama is a platform that allows the execution of LLMs in a safe and high-performance way. Its peculiarity is that it makes language models directly available to anyone on the user's personal device, thus avoiding the need for a connection to external servers. It is also highly compatible with Python, thus allowing easy interaction and processing of data and workflow.

d) ***Amazon Bedrock:*** [4]Amazon Bedrock is a service available through AWS that allows developers to access high-level generative AI models through a simple and scalable API. In this case, it was used to implement the Claude 3.5 Sonnet model with LangChain.

C. Projects analyzed

The study analyzed four open-source projects and one corporate project, all characterized by the Java programming language. The open source projects taken into consideration are the following: **ShardingSphere** [5], **OpenMrs**[6], **Crate**[7], **Teastore**[8] Furthermore, the company project, kindly provided by an Italian company [9], **Keyword**. We selected the above open-source projects since they have been previously analyzed for the detection and refactoring of CD smells [20]. This provided us with ground truths for the outcome of the LLM.

[2]Ollama
[3]LangChain
[4]Amazon Bedrockd
[5]ShardingSphere
[6]OpenMrs
[7]Crate
[8]Teastore

D. Preliminary interviews with developers (Phase 1)

An initial exploratory phase allowed us to gather opinions from experienced developers on the hypothetical effectiveness of the study. In particular, we used a plain application of LLMs on a few simple CDs occurring in two industrial projects that were provided by two Norwegian companies. In these cases, we didn't use specific techniques, but we simply asked the LLM to provide a refactoring suggestion and an explanation for such smells. We then discuss the results with the senior developers working on those projects. These preliminary results were promising, although with various shortcomings and unpredictability. However, the developers showed interest and appreciation, and this preliminary study increased our understanding of the state of LLMs and their potential for AS refactoring.

E. Experiments varying three variables (Phase 2)

A first comparative phase was conducted where only CDs of size 2 were considered (a total of 15 CD instances). This was done to avoid overloading the model and to understand if it was able to return valid refactoring suggestions for simpler cases. This phase focused on varying three main variables (application of well-known techniques to improve LLMs) which we describe in the following paragraphs:

a) ***Prompt:*** The first comparison involves differences in prompt detail. This means that the LLM receives different reasoning instructions and contexts of the dependency. This choice was made to explore the effectiveness of an LLM by providing more or less context (for example, the specific lines of code that generate the CD). This is an important point, as better prompts can yield better results [29].

A summary of both prompts can be seen in Figure 2 and examples of them can be found in the replication package [33]. In the first prompt(”*NotDetailed*”), the LLM is simply informed of the presence of a CD between two files, which are then passed to it within the prompt itself, carefully separated, so that the model can easily recognize the content of each file. The second prompt (”*Detailed*”) presents more details than the first. One of the prompting techniques used to enhance the initial prompt is role or role-play prompting ([30], [31]), meant to yield more contextually relevant outputs by customizing answers to align with specified role. First, the LLM is asked to behave like an expert architectural software developer in AS. Next, the LLM is asked to follow a series of rules to generate the answer called ”optimizers”. The use of optimizers consists in improving the prompt by making it structured and organized by breaking down the information to be passed to the LLM into steps. There is also the expression ”take a deep breath” which Yang et al. show to be a response

[9]Anoki

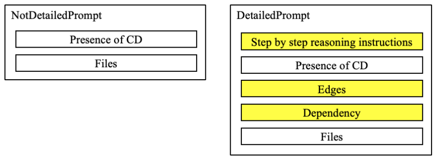


Fig. 2. Summary of the structure of the two prompts used, on the left the *NotDetailed* one, on the right the *Detailed* one.

optimizer [12]. Next, a set of rules are given that must be applied to generate the response. They are given in the form of steps to follow. These rules are:

”Use best practices for refactoring and the definition of the architectural smell to answer. Read the question carefully and ensure that you understand what the user is asking. Look for relevant information in the provided files and context. The answer can be as long as you want, try to be as specific as possible and include all the information you think is useful. Before finalizing your answer, take a deep breath and review your reasoning to ensure the answer is accurate and complete. Based on your reviewed reasoning, provide a clear answer to the user's question.”

Next, the LLM is informed about the presence of a CD between two files. Subsequently, the edges that make up the dependency graph of the CD are described verbally (in Fig. 2 this step is represented by the ”Edges” entry). For example, for the CD shown in Fig. 4, in the Edges section of the prompt, there is ”*FormField depends on Form. Form depends on FormField*”. Additionally, we give the LLM an explanation about why the two files depend on each other. This step was done by one of the authors, who viewed the CD through the dependencies graph and the files that generated it. So, the code elements for which the dependency is generated were collected and explicitly transcribed (in Fig. 2, this step is represented by the ”Dependency” box). This information is communicated to the LLM and, finally, the entire code of the two files is inserted and separated as in the *NotDetailed* prompt.

b) ***RAG****:* The next comparison involves the enhancement of the LLM with a RAG approach [19], where two central methods are employed: information retrieval and text generation. The strength of this technique lies in its ability to expand the domain knowledge of the pre-trained underlying model, enabling relevant information to be appended to the prompts, with the goal of obtaining fact-grounded answers. This information is efficiently retrieved and added to the context pool of the LLM, all of which avoids the costly efforts needed to retrain the model. There is an abundance of information available on CDs. Such, for examples, the various names with which they are classified in the literature and consequently multiple definitions, refactoring options and simple refactoring examples based on different types of scenarios. Furthermore, we find information about the types of CDs that can be encountered and how each of these causes problems within the codebase. This allowed us to provide the LLM with specific terminology, definitions, examples and documentation related to CD refactoring. In addition to the documentation above, the LLM was also provided with the files that cause the CD, as they can lead the LLM to notice further details in the code that can provide more useful context to remove a CD, for example documentation such as [21] [22] and papers on architectural smells such as [23] [24] (all the documents can be found in the replication package [33]). In technical terms, RAG integrates sophisticated text processing, embedding generation, and retrieval strategies in order to provide the LLM with the most relevant information from the context pool. This involves splitting the text documents into manageable chunks, followed by an embedding process which results in high-dimensional representations of the input documents, so-called embeddings, in a space where distances between elements translate to semantic relationships. Embeddings are stored in a vector database, allowing the retriever to search for and retrieve query-similar information.

c) ***LLM Type****:* We aim to compare the capabilities of local LLMs to cloud-based ones. These two differ according to a range of factors. More specifically, local LLMs appear to have more limited capabilities due to the fact that they are executed on the local machine, which often can be resource-constrained. While limited by the hardware capacity of the machine itself, local models maintain an unparalleled level of privacy by processing and storing data locally. On the other hand, cloud-based models boast important advantages in terms of both software and hardware, as they are executed on private infrastructures. The size of the models varies as well, local models typically featuring a lower number of parameters in order to accommodate the resource confines of the user's hardware. Among the various local models available, Qwen was chosen because according to a report [16] it performs better on code-related evaluation tasks datasets, including the most famous datasets MBPP[10], HumanEval[11] and EvalPlus[12], compared to local models of the same importance, such as DeepSeekCoder and CodeStral. As it can be seen from the report, Qwen was compared to much larger models, in fact note that Qwen is characterized by only $7B$ parameters while DeepSeek with $32B$ and Codestral with $22B$. As for the cloud-based model, Claude 3.5 Sonnet was chosen which, according to its technical report [17], appears to surpass other proprietary models such as ChatGPT4o and Gemini1.5.

In total, we have five answers computed for each analyzed CD: two answers are obtained by varying the prompt with the same LLM and without the use of RAG. Other two were obtained by varying the use of RAG and maintaining the same prompt and the same LLM. Regarding the last comparison, two answers were generated by varying the type of LLM but keeping the use of RAG and the Detailed prompt. The additional answer was then compared with the one already obtained in the second comparison. In Table I we summarize the three comparisons and, for each, we highlight which

[10]MBPP
[11]HumanEval
[12]EvalPlus

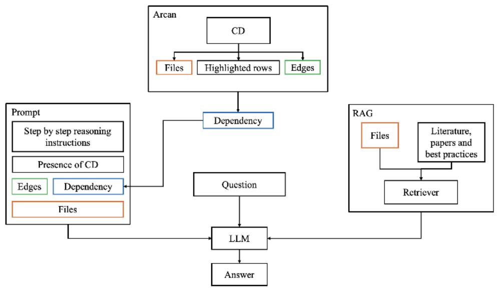


Fig. 3. Diagram showing the entire workflow structure which makes use of the *Detailed* prompt and the *RAG* technique.

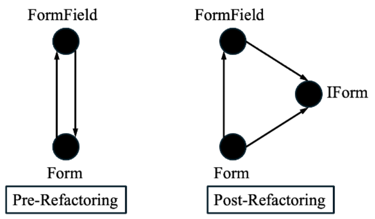


Fig. 4. The figure shows the dependency graph of the CD *o10*, before and after the refactoring proposed by the LLM.

techniques and LLMs were used for each obtained answer. Furthermore, a random test was added which consisted in proposing the same CD and relative comparisons to two different students. Therefore, 4 CDs were analyzed by pairs of different students. This allowed, for each of these 4 CDs, to have a double opinion and check if they matched, further validating the quality of the data.

Comparison	Answer 1	Answer 2
First	NotDetailed/Sonnet/NotRAG	Detailed/Sonnet/NotRAG
Second	Detailed/Qwen/RAG	Detailed/Qwen/NotRag
Third	Detailed/Qwen/RAG	Detailed/Sonnet/RAG

TABLE I
SUMMARY OF THE THREE COMPARISONS AND WHICH TECHNIQUES AND LLMS WERE USED TO OBTAINED THE TWO ANSWERS FOR EACH ANALYZED CD.

Before turning to the next phase, an example of a refactored CD from the LLM is shown, reporting the main points of the answer. In the LLM response proposed to break the CD in Fig. 4 we found the following short summary:

- *We introduced an IForm interface that Form implement.*
- *We kept the rest of the class Form unchanged.*
- *In the FormField class, we changed the type of form from Form to IForm, and consequently updated the return type of getForm() and parameter type of setForm() from Form to IForm. This removes the direct dependency on Form.*
- *The FormField class now deals with IForm objects instead of Form objects directly. This may require some casting in certain situations, but it breaks the CD.*

As for the code, the new `IForm` interface is shown below:

```
Step 1: Create a new interface called IForm

public interface IForm {
    // ...
}
```

For the `Form` class:

```
Step 2: Modify Form.java
public class Form extends ... implements IForm {
    // ... (keep existing code)
}
```

And finally for `FormField` class:

```
public class FormField extends ...  {
    // ... (keep other fields and methods)

    protected IForm form;

    public IForm getForm() {
        return form;
    }

    public void setForm(IForm form) {
        this.form = form;
    }

    // ... (keep other methods)
}
```

This answer was generated using the *Detailed* prompt, Claude 3.5 Sonnet and the RAG technique. The answer was rated positive because it presents a correct refactoring proposal, a step-by-step reasoning concluded with a concise summary, and the relevant code in the summary. As we can see from Fig. 4, the two classes now depend on the interface since Form implements it and `FormField` use `IForm` objects instead of directly calling the concrete `Form` class.

Replication package The replication package [33] contains the RAG documentation, the prompts and questions used, some hallucinated answers, the code and the answers of some of the refactored CDs. The responses are named with a three-character code. The first character indicates the prompt used (*1* NotDetailed, *3* Detailed), the second, whether RAG was used (*N* NotRAG, *R* RAG) and the third the model used (*S* Sonnet, *Q* Qwen).

F. Qualitative evaluation and interviews with Master's students (Phase 3-4)

The validation phase of these results consisted of a first individual evaluation carried out by one of the authors, and a subsequent one with 10 interviews with master students. The individual evaluation was performed by selecting the answer that was closer to a refactoring previously performed on the same CD ([20]) among the generated during the three comparisons (see section III.E and Table I).

For the answers obtained with Qwen, a total of 10 answers were generated each time. This was due to the fact that the answers of this model were affected by hallucinations [33] and sometimes the model was able to grasp the context (specifically, it stated that it was not aware of the files to be processed even though these were correctly provided). Therefore, after generating the ten answers, we chose the most relevant and understandable one and used it for the analysis.

As for the answers of Claude Sonnet, after some tests, where it was noted that it remained constant in the type of answer generated and always managed to grasp the data proposed, it was decided to generate a single answer for each scenario.

Once we generated the various refactoring suggestions according to the variation of our studied variables (prompt, RAG, and LLM, see section III.E and Table I), we showed such suggestions during interviews with master students with advanced knowledge of CD definition and refactoring and the negative impact that it can cause to the codebase. For each CD presented to the students, we showed the code and the reasons why a CD occurred between the two files. Once the context was understood, we presented the refactoring suggestions organized in 3 pairs according to the studied variable (see Table I). This approach ensures that, for each variable, we obtain a clear preference from each interviewee regarding which suggestion they considered better. The interviewee had time to view each suggestion and understand the proposed refactoring. At this point, we asked the interviewee which of the two suggestions they preferred and the reasons that led them to such choice. Furthermore, we asked a vote from 1 to 4 on two factors:

- **Comprehensibility**: indicates how clear the answer is, more specifically, whether it was difficult to understand the type of refactoring proposed. Hypothetical linguistic and contextual hallucinations of the LLM must also be taken into account. Finally, the organization of the suggestions and the code shown are taken into consideration.
- **Usefulness**: this value mainly takes into account the practical usefulness of the proposed refactoring of the CD. This also includes the alternative when the actual code refactoring proposed by the LLM is not directly applicable, but the explanation is sufficiently useful to direct the developer towards a correct solution.

The rating ranges from 1 to 4 so that there are no intermediate ratings that allow the interviewee to remain neutral. In fact, if 4 is the best rating and 1 the worst, 2 is unbalanced negatively and 3 is unbalanced positively. Since there is no rating between 2 and 3, the interviewee will always be forced to take a positive or negative position.

The total number of CD presented to the students is 15 and is reported in Table II. The four CDs analyzed by two different students are *ss6*, *o13*, *t1* and *c9*. Since there were opinions from two students for these four CDs, before reporting them together with the others, it was checked that the opinions for each comparison of each CD matched. In fact, if the two students had stated contrasting answers, these CDs would not have been considered in the final result.

G. Experimenting on more complex CDs (Phase 5-6)

Following interviews with students, we decided to increase the difficulty of the CDs to be refactored. As the size of the CD increases, the amount of code and information necessary to reason about the refactoring also increases. Consequently, an LLM must receive this information as well. However, the greater the context, the more difficult often is for the LLM to propose a correct and precise answer, as additional context can also introduce noise. In fact, when trying to apply the methodology used for size 2 CDs on larger CDs, the LLM Qwen was unable to give a meaningful response and was completely hallucinated, while Claude Sonnet responded in a very general way by often avoiding proposing solutions with an explicit lead.

Project	IDs
ShardingSphere	ss6, ss2, ss3, ss5
Crate	c12, c7, c6, c9
Openmrs	o13, o10, o9, o12, o11, o8
TeaStore	t1

TABLE II
THE LIST OF SIZE 2 CDS PROPOSED DURING INTERVIEWS WITH MASTER'S STUDENTS IN PHASE 4.

In this part of the study, based on the better results obtained in phases 2-4, we chose to use the LLM Claude Sonnet 3.5, applying the *RAG* technique and making use of the *Detailed* prompt. This structure is shown in Fig. 3. This figure shows how the workflow is organized from the various inputs given to the LLM, to its answer. At the top is the Arcan box from which the information on the analyzed CD is obtained. On the left, is the prompt box, already extensively explained in phase 2. On the right is the RAG box, which provides the LLM with documents and information useful for improving the content of the response.

The problem of the management of greater context was split into two subproblems which are managed by two independent models, powered by the same LLM:

1) **Edge to break**: for this initial study, smells characterized mainly by the circular shape were considered, since the breaking a singular arc allows breaking the entire CD. In this first phase, the same model is applied as in Fig. 3. The question is changed, no longer requiring a refactoring of the code, but rather questions which of the dependencies present in the prompt are the best to break. This way, the LLM is not overloaded with the request to report a refactoring that includes all the files of the dependency. Instead, it identifies a dependency that is weaker and easier to break. Although not required, the response includes actionable advice for breaking the addiction. This step is necessary to identify which part of the code needs to be visited to remove the dependency. In this way the LLM does not focus on providing refactored code but on the analysis of what the CD presents.
2) **Refactoring proposal**: at this point, the second LLM takes care of generating a refactored code. In this case, however, the prompt contains the information about the dependency that is intended to be broken and only the code of the two files involved. At this point, the question provided to the second LLM asks to break the specific dependency by applying the suggestion proposed in the answer obtained previously. This step allows the LLM

to focus on correcting the code, with the sole purpose of applying the advice obtained. This does not involve the analysis of additional code that is not necessary for correcting the CD.

The experiments conducted for the generation of data to be validated later were based on a total of 8 CDs, 4 of size three and 4 of size four. One CD of size three is from the ShardingSphere project, one CD of size four is from the Crate project, and the remaining ones are from the Keyword project.

H. Individual evaluation, Interviews with Master students and developers (Phases 7-8-9)

The validation phase consisted of an individual evaluation by one of the authors and a subsequent one with a series of interviews with master students and two developers. The personal evaluation, for the two CDs not belonging to the Anoki project, was done by comparing them with the refactorings of such smells obtained in a Master's thesis [20]. So, for each of these two CDs two interviews were conducted, for a total of four interviews with master's students. This allowed us to obtain opinions from two different students for each CD. These opinions were then compared to see if both students agreed or not. The interview initially focused on the analysis of the dependency graph. Then, after having looked at the code and what the CD entails, the student was asked which dependency he would break and how, to remove the CD. Once this was done, the refactoring proposed by the LLM was shown. At this point we reasoned together with the student to understand if what the LLM proposed was what he himself had hypothesized to apply. If not, the student was asked if the refactoring obtained by the LLM was correct. Finally, as for the dependencies of size 2, a graded was asked on the Comprehensibility and Usefulness of the answer. For the CDs identified in Keyword project, first a refactoring was personally proposed and then compared with the one generated by the LLM. Subsequently, two interviews were conducted with two developers who personally know the code base. For the CDs analyzed with the developers, the same interview approach was applied. The double interview method was not applied to the same CD.

IV. Results

In Table III, we outline the preferences given by the interviewed Master students for the 15 studied CDs. In each comparison, we studied one specific variable: prompt, RAG and LLM type. Note that the total responses for the first comparison is fourteen and not fifteen because one of the participants was unable to express a motivated preference.

A. RQ1. To what extent does providing the LLMs with more detailed prompt give better or worse suggestions to refactor a CD architectural smell?

Looking at the results obtained at the top of Table III, the Detailed prompt clearly obtains more satisfactory results. This confirms the essential need to provide context for dependency resolution, probably because if the reason for the presence of the CD is not specified, the LLM is forced to take into account the entire file, failing to focus on the problematic code. In fact, during the interviews, it was identified that through the use of the NotDetailed prompt, the LLM often tries to propose to create a new class, which is often not motivated appropriately, and the introduction of the latter is not as easily applicable as an interface. Furthermore, the answers are clearer and more concise, probably because the LLM is given a specific focus based on the information provided via the prompt.

Comparison	**NotDetailed**	**Detailed**
Prompt	3	**11**
Mean assessment	-	C: 3,18 / U: 2,91

Comparison	**NotRAG**	**RAG**
Use of RAG	4	**11**
Mean assessment	-	C: 3 / U: 2,81

Comparison	**Qwen2.5-Coder**	**Claude 3.5 Sonnet**
Local or Cloud LLM	6	**9**
Mean assessment	-	C: 3,22 / U: 2,89

TABLE III
THE RESULTS OF THE THREE COMPARISONS OBTAINED DURING THE INTERVIEWS WITH THE MASTER STUDENTS.

In conclusion, a more detailed prompt provided better refactoring suggestions. This is supported by the higher preference (78% of the participants) for the answers given by the enhanced LLMs with detailed instructions.

B. RQ2. To what extent does using local LLMs give better or worse suggestions to refactor a CD architectural smell than using a cloud-based model?

The results obtained in Table III, regarding the comparison between local and cloud-based LLMs, show how there is no clear agreement on which performs best. Of 15 total CDs, 9 Sonnet-generated responses were preferred, while 6 Qwen-generated responses were preferred. A drawback of Qwen, however, is that, for each response generated, it was necessary to generate a total of 10, discard the completely hallucinated ones manually and choose the best of the remaining ones considering the coherence between the context of the question and the one of the answer. However, after the filtering, some of Qwen's answers were preferred by the interviewees as they proposed an effective refactoring method and, in some cases, even clearer and more concise than Sonnet. This implies that the output of Qwen might work better in practice, but only if refined.

In conclusion, although more participants preferred the cloud LLM-provided responses in terms of usefulness, our findings highlight the potential for refining local LLMs, as close to half of the participants (40%) preferred the responses given by the local model.

Project	ID	Size	Comprehensibility	Usefulness
ShardingSphere	ss1	3	4 (4, 4)	4 (4, 4)
Crate	c10	4	4 (4, 4)	3,5 (4, 3)
Keyword	k20	3	3	3
Keyword	k47	3	4	4
Keyword	k05	3	3	3
Keyword	k12	4	4	3
Keyword	k81	3	3	4
Keyword	k61	4	4	3
Keyword	k69	4	3	3
Avarage			3,56	3,39

TABLE IV
LIST OF SIZE 3 AND 4 CDS PROPOSED DURING INTERVIEWS WITH MASTER'S STUDENTS AND DEVELOPERS WITH RESPECTIVE GRADES OF COMPREHENSIBILITY AND USEFULNESS.

C. RQ3. To what extent does the use of Retrieval Augmented Generation (RAG) improve the suggestions given by the LLMs on how to refactor a CD Architectural Smell?

As we can see from the results reported in Table III the answers generated through the use of the RAG Technique were clearly preferred: 11 answers were rated better out of a total of 15. The reasons that led students to prefer this can be summarized in a series of recurring concepts, analyzed from their explanations:

- **Better organization**: The responses generated through the RAG technique appear to have a more refined and comprehensible structure.
- **Better refactoring solution**: the answers generated through the RAG technique appear to propose more suitable refactoring solutions. For example, sometimes such refactoring suggestions introduce fewer elements (only one interface, instead of both an interface and a class, providing a better logic structure). In some cases, however, the RAG answer was preferred because the one generated without the RAG was not correct at all or completely inconclusive.
- **Step by step reasoning**: in some cases, students have noticed that the answers generated with the RAG technique are more understandable because they report a step-by-step reasoning. This allowed them to follow the reasoning without being overwhelmed by a full solution without specific motivations.

In conclusion, using the RAG technique led to a substantial improvement in the refactoring suggestion given by the LLM, with 73% of participants preferring the results of the RAG-enhanced LLM due to better logic structure and refactoring suggestions.

D. Overall RQ. To what extent does the use of LLMs help to refactor a CD architectural smell?

To answer this question, we have to consider the results outlined for the previous RQs together with the results obtained when increasing the complexity of the analyzed CDs. As we can see in Table III, the value averages of Comprehensibility and Usefulness are clustered around 3, and so positively skewed towards the highest degree of satisfaction,

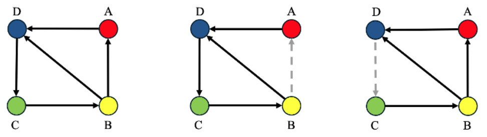


Fig. 5. From left to right: the dependency graph of the CD before refactoring. The dependency graph with the dashed edge to break according to the developer. The dependency graph with the dashed edge to break according to the LLM.

4. In addition, as reported in Table IV, the results of the additional interviews, based on CDs of size 3 and 4, gave even higher results for Comprehensibility and Usefulness. This means that, by refining the approach, the LLMs give better results.

The data in Table IV confirm that on average the solution is highly understandable. The only detail that was not particularly appreciated by one of the developers, was the truncation, in the generated suggestions, of the code parts and the replacement with suspension points. However, this detail is not considered in the final conclusions as several students appreciated the conciseness of this truncation method, making the key changes of the refactoring more understandable. Furthermore, we notice a total lack of hallucinations and a total coherence between the explanatory part and the suggested code. This is probably due to the fact that the context was made even more specific with the additional techniques, even though the amount of information is the same, but administered in a more organized way. Furthermore, the use of two sequential models, which divide the responsibilities, are able to obtain a more accurate result. As for the Usefulness, it is recognized by the interviewees that the applied method obtains results that are very useful to correct the CD smell.

As for the four interviews with students, all of them, before viewing the LLM's response, proposed breaking the CD by modifying the same file that the LLM proposes to refactor, therefore acting on the same dependency arc. While, as far as developers are concerned, it is necessary to specify some details. In fact, considering the 7 CDs proposed to them, in 4 CDs the dependency arc to be broken proposed by the developer and the LLM coincides. The last three scenarios analyzed are reported in detail below. Here, the dependency arc to be broken to solve the CD proposed by the LLM and the one proposed by the developer do not coincide. The first one, $k69$ in IV, the CD is characterized by a dependency loop on all 4 files (*A*-*B*-*C*-*D*) and a smaller one that characterizes only 3 of the 4 total (*B*-*C*-*D*) that can be seen in Fig. 5. Note that the removal of the arc suggested by the developer does not correct the entire scenario but only the general loop. After actually showing the solution proposed by LLM, which create a new interface which is implemented by class C and called by class D, the developer states "I had not considered this solution, it gets a greater decomposition of the CD. It also breaks the smaller loop between files B, C and D". In the second one, $k81$ in IV, the developer suggests, without considering the presence of the CD, the correction of an Enumeration file because its non-static nature is not appreciated by the developer considering the type of responsibility it covers. Therefore, in this case, the developer's interest is more focused on correcting another type of error that does not concern the presence of the CD. In the last scenario, $k05$ in IV, the developer suggests breaking an arc that he thinks is easier and safer to remove. After seeing the LLM's suggestion, he thinks the proposed refactoring is correct, but prefers his own approach since the file he thought needed fixing was more disorganized.

In general, both developers recognize the usefulness of LLM in helping to understand and remove the CDs in the code. In fact, both developers state that AI allows to have an initial vision of the problem and a first approach that can be developed and integrated with the rest of the codebase.

Furthermore, to both of the developers were shown, without making it explicit, a particular scenario. In fact, two of the analyzed dependencies (for each developer) present a shared dependency arc. This means that this two CDs have parts of the codebase in common. Both noted that it would be particularly useful to inform the LLM of the presence of this shared arc between CDs in order to suggest a refactoring that can solve both CDs in one go. In both cases, the LLM was provided with information about the two CDs (as done in the other tests) and mentioned that they share "some files and dependencies". In both cases the LLM proposed a simple and effective refactoring for both shared CD scenarios. This allows us to state that this automatic refactoring approach can be expanded and made more effective in more important CD. The fundamental idea is to communicate with extreme precision the critical points and divide the responsibilities.

In conclusion:

- The use of LLMs to give refactoring suggestions for CD of size 2, 3 and 4, gives generally satisfactory results, making it a promising approach to be adopted in the future to support developers.
- The use of techniques to give additional context (such as a detailed prompt and RAG) to the LLM is key to improve the suggestion.
- The use of a multi-agent approach helps refining the instructions to generate a satisfactory answer for complex CDs.

V. Threats to validity

This study is intended as a first exploration towards automatic refactoring of CD smells through LLM. The study has some limitations, first of all, a small number of CD instances have been analyzed. This could lead to the analysis of scenarios that are too specific and do not include particular cases. However, we tried to analyze different scenarios, varying in 5 different projects. Furthermore, metrics such as Comprehensibility and Usefulness have been introduced that can be extremely subjective, therefore not defining a standard. By providing double opinions on some CDs during interviews with students, we tried to limit the subjectivity of such statements. It should also be noted that the comparisons analyzed by the students were structured in such a way as to reduce the amount of elements to be evaluated as much as possible (since the method of verifying the answers through interviews involves a long time). In fact, there are no tests on the behavior of Qwen through the use of the two different prompts and comparison tests between Claude 3.5 Sonnet and Qwen without the use of RAG. A final consideration concerns the use of Qwen, as already reported, during the use of this model a total of 10 were generated for each answer. This led, by one of the authors, to a manual choice of the answer to be analyzed during the interviews with the students. However, the eliminated answers were those where the template stated that it had not been made available for files that needed refactoring or where the context of the answer did not fully respect that of the question.

VI. Discussion and Concluding Remarks

In this paper, we explored the use of LLMs to support the refactoring of CDs smells. We explored different techniques, and we interviewed students and developers to assess the usefulness of the given suggestions.

According to our assessment, the results can be considered promising and worth further exploration in future research. For small CDs (size 2), the use of techniques such as detailed prompt and RAG gives generally acceptable results. When examining more complex CD (size 3 or 4), the simple use of LLMs, even when enriched with such techniques, do not work. We find that the key is to give the LLM not just more context, but more precise context. This provides better refactoring suggestions for more complex CDs. We could argue that relying on the user to give such input (for example, specifying which dependency is better to remove) would burden the user and would require the user to have already solved the problem. However, with the use of a multi-agent approach (in this case, two LLM-based agents with specific roles), seems to perform well and provide very good refactoring suggestions.

In practice, this means that developers would probably not receive satisfactory suggestions by just using a simple LLM to aid refactoring a CD. However, the combination of a tool to recognize CDs (such as Arcan, as experimented in this study), a detailed prompt, and RAG would give very good results for small CDs. When increasing the complexity, though, a multi-agent approach is needed. Although these are approaches that can be implemented, we don't see them as directly and easily accessible to software developers. However, the use of LLMs can be implemented as additional features on top of existing static analysis tools. Given the size of a dependency to be solved, a more specific approach can be selected to provide refactoring suggestions.

When it comes to selecting a specific LLM, although the cloud-based one seems to perform slightly better, the choice is not obvious. In some cases, the local LLM would still perform well enough and was even preferred by the interviewees. Despite some drawbacks, with further refinements we envision that local LLMs can be used for the task. This can be a deal-breaker for some organizations that would not want to have their data processed by the LLM remotely. More and more powerful models are released every day, so it is possible that local LLMs will be available to be used in the future.

Further research should also address CDs of higher complexity. We have started with the base cases (size 2) and increased the complexity incrementally to size 3 and 4, but it is possible that, by increasing it again, further techniques need to be explored to tackle larger CDs. In addition, more CD instances, projects and interviews, especially in industrial contexts, should be considered and carried out to obtain additional evidence. Moreover, other architectural smells, such as Unstable Dependency and God Component can be analyzed to extend our study. It would be also interesting to analyze how the LLM can handle different types of smells together, and consider also the case where some kind of dependency or correlation may exist among the smells. Furthermore, other techniques can be explored, for example, training a specific LLM to suggest refactoring of a specific smell etc. Finally, in this paper, we explored LLMs as aid to give refactoring suggestions, but the next step is to explore the automatic refactoring of CDs.

REFERENCES

[1] Q. Guo et al., ”Exploring the Potential of ChatGPT in Automated Code Refinement: An Empirical Study” Sep. 15, 2023, arXiv: arXiv:2309.08221. Acc: Sep. 2024. Available: http://arxiv.org/abs/2309.08221

[2] A. Shirafuji et al., ”Refactoring Programs Using Large Language Models with Few-Shot Examples” in 2023 30th Asia-Pacific Software Engineering Conference (APSEC), Dec. 2023, pp. 151–160. doi: 10.1109/APSEC60848.2023.00025.

[3] F. A. Fontana et al., ”Arcan: A Tool for Architectural Smells Detection” in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg: IEEE, Apr. 2017, pp. 282–285. doi: 10.1109/ICSAW.2017.16.

[4] D. Sas et al, ”Investigating instability architectural smells evolution: An exploratory case study”. In 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019, Cleveland, OH, USA, September 29 - October 4, 2019, pages 557-567. IEEE, 2019.

[5] R. Capilla et al., ”Detecting architecture debt in micro-service open-source projects”. In 49th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2023,September 6-8, 2023, pages 394–401. IEEE, 2023.

[6] F. A. Fontana et al., ”Impact of architectural smells on software performance: an exploratory study”. In Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, EASE 2023, Oulu, Finland, June 14-16, 2023, pages 22–31. ACM, 2023.

[7] R. Kazman et al., ”A Tool to Address Cybersecurity Vulnerabilities Through Design” Carnegie Mellon University. 29-Feb-2016. Available: https://insights.sei.cmu.edu/blog/a-tool-to-address-cybersecurity-vulnerabilities-through-design/.

[8] T. D. Oyetoyan et al., ”Circular dependencies and change-proneness: An empirical study”. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, Mar 2015, Montreal, Canada. ff10.1109/SANER.2015.7081834ff. ffhal01203525f

[9] H. Farsi et al., ”A Graph-based Solution to Deal with Cyclic Dependencies in Microservices Architecture” 2022 9th International Conference on Future Internet of Things and Cloud (FiCloud), Rome, Italy, 2022, pp. 254-259, doi: 10.1109/FiCloud57274.2022.00042.

[10] U. Azadi et al., ”Architectural Smells Detected by Tools: a Catalogue Proposal” in 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), Montreal, QC, Canada: IEEE, May 2019, pp. 88–97. doi: 10.1109/TechDebt.2019.00027.

[11] D. Sas et al., ”The Perception of Architectural Smells in Industrial Practice” IEEE Softw., vol. 38, no. 6, pp. 35–41, Nov. 2021, doi: 10.1109/MS.2021.3103664.

[12] C. Yang et al., ”Large Language Models as Optimizers” Apr. 15, 2024, arXiv: arXiv:2309.03409. doi: 10.48550/arXiv.2309.03409.

[13] B. Zhang et al., ”Copilot-in-the-Loop: Fixing Code Smells in Copilot-Generated Python Code using Copilot” Aug. 21, 2024, arXiv: arXiv:2401.14176. doi: 10.48550/arXiv.2401.14176.

[14] C. E. Jimenez et al., ”SWE-bench: Can Language Models Resolve Real-World GitHub Issues?” Apr. 05, 2024, arXiv: arXiv:2310.06770. Acc: Sep. 2024. Available: http://arxiv.org/abs/2310.06770

[15] S. Herold et al., ”Using Automatically Recommended Seed Mappings for Machine Learning-Based Code-to-Architecture Mappers” Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, Tallinn, Estonia. doi 10.1145/3555776.3577628.

[16] B. Hui et al., ”Qwen2.5-Coder Technical Report” Sep. 24, 2024, arXiv: arXiv:2409.12186. doi: 10.48550/arXiv.2409.12186.

[17] Anthropic, ”Claude 3.5 Sonnet Model Card Addendum”. Available: https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf

[18] R. C. Martin, ”Agile Software Development: Principles, Patterns, and Practices”. 2003. Prentice Hall PTR, USA.

[19] P. Lewis et al., ”Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”, Apr. 2021. URL http://arxiv.org/abs/2005.11401. arXiv:2005.11401 [cs].

[20] P. Bacchiega, ”Analysis of the impact of architectural smell refactoring on energy consumption”, Master Thesis of Computer Science, University of Milano Bicocca, October 2024.

[21] Arcan, ”Architectural Smells” Arcan Documentation. Available: https://docs.arcan.tech/2.9.0/architectural_smells/. [Acc: Sep. 2024].

[22] Arcan, ”Refactoring” Arcan Documentation. Available: https://docs.arcan.tech/2.9.0/refactoring/. [Acc: Sep. 2024].

[23] G. Suryanarayana et al., ”Refactoring for software design smells: managing technical debt”. Amsterdam; Boston: Elsevier, Morgan Kaufmann, Morgan Kaufmann is an imprint of Elsevier, 2015.

[24] F. A. Fontana et al., ”A Study on Architectural Smells Prediction” in 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Kallithea-Chalkidiki, Greece: IEEE, Aug. 2019, pp. 333–337. doi: 10.1109/SEAA.2019.00057.

[25] H. Mumtaz et al., ”A systematic mapping study on architectural smells detection” Journal of Systems and Software, vol. 173, p. 110885, Mar. 2021, doi: 10.1016/j.jss.2020.110885.

[26] R. C. Martin, ”Granularity”. Available: https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/granularity.pdf. [Acc: Sep. 2024].

[27] R. C. Martin, ”The Dependency Inversion Principle”. Available: https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/dip.pdf. [Acc: Sep. 2024].

[28] D. Sas et al., ”An architectural technical debt index based on machine learning and architectural smells” Oct. 17, 2023, arXiv: arXiv:2301.06341. doi: 10.48550/arXiv.2301.06341.

[29] S. M. Bsharat et al., ”Principled instructions are all you need for questioning llama-1/2, gpt-3.5/4” 2023, arXiv: arXiv:2312.16171.

[30] S. Schulhoff et al., ”The Prompt Report: A Systematic Survey of Prompting Techniques” 2024, arXiv: arXiv:2406.06608.

[31] A. Kong et al., ”Better zero-shot reasoning with role-play prompting” 2023, arXiv: arXiv:2308.07702.

[32] Forbes, AI Writes Over 25% Of Code At Google What Does The Future Look Like For Software Engineers? https://www.forbes.com/sites/jackkelly/2024/11/01/ai-code-and-the-future-of-software-engineers/ [Acc: Jan. 2025]

[33] G. Pandini. (2025). ”An exploratory study on architectural smell refactoring using Large Languages Models” (Version 1.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.14745953