2025 IEEE/ACM 4th International Conference on AI Engineering – Software Engineering for AI (CAIN) | 979-8-3315-0219-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/CAIN66642.2025.00049

Task decomposition and RAG as Design Patterns for LLM-based Systems

Orlando Marquez Ayala
ServiceNow
orlando.marquez@servicenow.com

***Abstract*—AI technologies are moving rapidly from research to production. Compared to traditional AI-based software, systems employing Large Language Models (LLMs) are more difficult to design due to their scale and versatility. This makes it necessary to document *best practices*, known as design patterns in software engineering, that can be used across LLM-based applications.**

While Task Decomposition and Retrieval-Augmented Generation (RAG) are well-known techniques, their formalization as design patterns for LLM-based systems benefits AI practitioners. These techniques should be considered not only from a scientific perspective, but also from the standpoint of desired software quality attributes such as safety and modularity. This will help bridge the gap between AI and software engineering as those fine-tuning or prompting LLMs will be aware of the impact that modern techniques have on the overall system.

I. Background

Due to current limitations in Large Language Models (LLMs), such as their black-box nature, building systems leveraging their capabilities is challenging. Integrating AI models into software already adds a new set of complex design choices to software engineering [1], yet this complexity has increased due to an LLM's expanding abilities to generate almost any kind of output. For instance, whereas in the past AI models added classification or regression capabilities to software, nowadays systems include more sophisticated features such as summarization and code generation.

There is a rich body of literature on software engineering for Machine Learning (ML), ranging from case studies to surveys drawn from academic and gray literature, some of which contribute a list of design patterns for AI-based systems [2]. A limitation of literature reviews is that they necessarily lack detail, describing design patterns at a high-level, divorced from the context in which they are applied. Moreover, most of the existing list of ML design patterns date from the pre-generative AI era, when AI components were simpler.

Generative AI requires adapting existing design patterns and adding new ones, as problems specific to LLMs such as hallucination (safety) or large number of output tokens (scalability) can affect software systems in new, unexpected ways. As the typical process to build AI-based systems still sees a rift between scientists and engineers, where the former develop the models and the latter bring the models into production [3], formalizing well-known LLM techniques as design patterns will help AI practitioners. Our aim is to shed light on the advantages and disadvantages, in terms of software quality attributes, of current state-of-the-art AI.

II. Approach

Our proposal derives from our experience building a real-world enterprise application[1] that relies on an LLM to generate a workflow in JSON format given a user textual instruction. While LLMs enable delivering rich and time-saving functionality, integrating them into commercial software for thousands of end-users requires careful scientific experimentation and sound software engineering.

To ship an acceptable solution, we needed to address typical generative AI challenges such as an LLM's propensity to hallucinate, high latency resulting from large number of input and output tokens, and the need to include data from the environment in the LLM's output. To this end, we leveraged two AI techniques: Task Decomposition and Retrieval-Augmented Generation (RAG), because they helped us reduce, albeit not completely remove, the risks associated with these challenges.

Task Decomposition is an application of *divide and conquer* to the AI domain, which breaks an ML task into sub-tasks that can be more easily solved. RAG is a well-known technique to allow an LLM to interact with data from its environment.

While the first objective was to devise a model/pipeline that would provide good output quality to the user, a solution could only be shipped if it met desired software quality attributes. Throughout the AI development cycle, from data labeling to deployment, we noted the advantages and disadvantages of these two techniques, attempting to generalize from our particular experience and use case to a blueprint solution in the form of a *design pattern*.

For example, since modularity tends to be desirable, does the technique contribute or hinder modularity in the overall system? Appealing user interfaces and user experiences help increase customer adoption, then does the technique enable better user engagement? Lastly, since LLM-based systems tend to be difficult to test due to the large range of possible input, how does this technique help the testability of the system?

III. Discussion

A. Task Decomposition

Breaking an ML task into simpler sub-tasks is beneficial when the task is too complex to solve with one LLM call, as smaller problems can be more easily tackled, thereby improving the functional correctness of the system. Another

[1] https://www.servicenow.com/docs/bundle/xanadu-build-workflows/page/administer/flow-designer/concept/flow-generation.html

979-8-3315-0219-5/25/$31.00 ©2025 IEEE
DOI 10.1109/CAIN66642.2025.00049

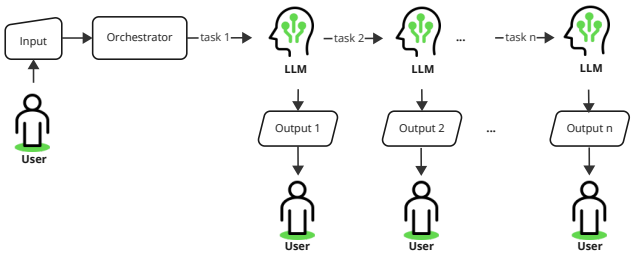
key benefit is improved user engagement as the output of sub-tasks, or intermediary output, can be shown to users without having to wait for the entire generation to complete. In fact, this technique is suitable for human-in-the-loop as users can correct the intermediary output before calling the remaining sub-tasks. An example of such task is generating code for a complex class: The AI model can first generate the signatures of the class functions, and then generate their content.

Other enhanced software quality attributes are modularity, testability, and analysability. A sub-task can be shipped as a module that includes its own pre- and post-processing, and its prompt and input can be modified independently. Sub-tasks can be individually tested, each having its own set of evaluation metrics and dataset, allowing effort to be concentrated on the under-performing sub-tasks. Once the system is deployed, problems can be diagnosed at the sub-task level, resulting in easier debugging.

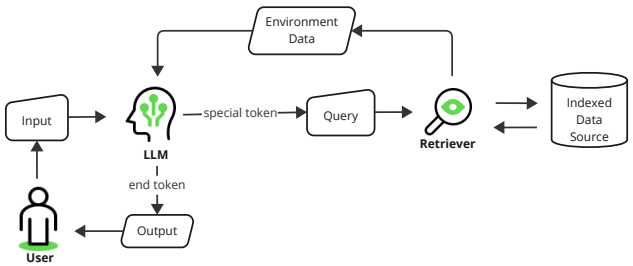
The disadvantages of Task Decomposition are additional overhead to orchestrate the sub-tasks and perhaps lower scalability due to a higher number of LLM calls. This technique requires another component to divide the task, collect their input and unify their output.

Figure 1a shows a sample structure of Task Decomposition. An implementation question is how to orchestrate the sub-tasks: in parallel or sequential? The answer depends on specific system considerations such as whether partial output can be shown to users and whether the input of a sub-task depends on output from previous sub-tasks.

Chain-of-Thought prompting and Multi Agent AI are two other approaches to break a complex ML task into sub-tasks.



(a) Task is decomposed into sequential sub-tasks to show results to users incrementally.



(b) LLM decides when to request for data based on outputting special tokens, until an end token is reached; also known as adaptive RAG.

Fig. 1: Sample structures of Task Decomposition and RAG.

B. Retrieval-Augmented Generation

RAG is popular among AI practitioners because it provides many benefits. Suggesting facts to an LLM during generation can reduce hallucination (safety). To simplify the ML task, it is desirable to separate *how* the task is solved from *what* data is needed to solve it (modularity). LLM-based systems tend to be black boxes; showing the retrieval results gives users more visibility on the system's inner workings (self-descriptiveness).

An important requirement of generative AI applications is that the LLM must interact with the environment and leverage the information available in it. This results in improved functional correctness as the LLM is not limited by the knowledge available in its training data, which is encoded in its model weights. While RAG does allow interoperability with data sources such as databases, there is no guarantee that the LLM stops hallucinating or that it uses appropriate information in its output. In addition, RAG introduces other AI and non-AI components into the system that can cause additional latency and maintainability: an embedding model to encode text and a retriever module that may require complex pre-processing (e.g., chunking) and post-processing (e.g., re-ranking).

When errors in the output arise, RAG allows a diagnostic of whether the LLM made an error due to lack of AI capabilities, or due to inappropriate environment data available to it. System safety is improved due to reduced hallucination but security risks are introduced as the data sources used during generation can be targets of security attacks.

Figure 1b shows an example where an LLM requests environment data from a retriever module only when it outputs a special token. Tool Usage is another approach to allow LLMs to access external sources of data. This approach is more flexible as the LLM can call any available API, but it may introduce more security risks as arguments passed to these function calls may result in unauthorized output [4].

IV. Conclusion

Based on our practical experience, we propose a formalization of two well-known AI techniques, Task Decomposition and RAG, as design patterns for LLM-based systems. We discuss their advantages and disadvantages in terms of software quality attributes, aiming to help AI practitioners consider their impact on the software engineering of the system. As LLMs are deployed more and more into larger applications, we judge it imperative to extend this analysis to other common state-of-the-art AI techniques such as Multi Agent AI and Tool Usage.

References

[1] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, "How does machine learning change software development practices?" *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1857–1871, 2021.

[2] L. Heiland, M. Hauser, and J. Bogner, "Design patterns for ai-based systems: A multivocal literature review and pattern repository," in *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, 2023, pp. 184–196.

[3] S. J. Warnett and U. Zdun, "Architectural design decisions for the machine learning workflow," *Computer*, vol. 55, no. 3, pp. 40–51, 2022.

[4] Z. Wu, H. Gao, J. He, and P. Wang, "The dark side of function calling: Pathways to jailbreaking large language models," 2024. [Online]. Available: https://arxiv.org/abs/2407.17915