# STRUCTURED REVIEW ON RAG- AND MULTI-AGENT FRAMEWORKS – PART II: APPLICATION-BASED ASSESSMENT

*Md Monsur Ali, Abdullah Al Foysal, Siddarth Venkateswaran, Ronald Böck*
*Genie Enterprise Deutschland GmbH*
*{firstname.surname}@genie-enterprise.com*

**Abstract:** Multi-Agent Frameworks, Agentic Systems, and related approaches are currently emerging, also in terms of applications. They are mainly based on Large Language Models, especially in terms of conversational agents. In Part I of our contribution, we considered Multi-Agent Frameworks in a theoretical way and provided an overview of state-of-the-art methods. In addition, the current paper presents an application-based assessment of multiple frameworks, covering a wide range of characteristics. As we were interested in the capability, scalability, and complexity of such frameworks in real-world settings, we applied the GenieReader, a tool being designed to address management and extraction of information from complex documents, as application. The frameworks were integrated into the current workflow and tested on data taken from the insurance domain. As a baseline, we used the GenieReader with a Vanilla RAG setting. Finally, we derived multiple hands-on take home messages for the use of Multi-Agent Frameworks.

## 1 Introduction

This manuscript is the second part of our contribution to review Multi-Agent Frameworks (MAFs), considering also Retrieval-Augmented Generation (RAG) and Multi-Agent Systems (MASs). In the first part, being included in the current proceeding's edition as well, focused on an overview of approaches and a systematic review. In the current paper, we selected LangGraph, CrewAI, and SmolAgents for an application-based assessment, based on their structure, usability, and relevance to multi-agent coordination. The key contribution of the current work is to test these frameworks in a real-world application, being intended as a representative of common task, nowadays handled by Large Language Models (LLMs) and MASs.

In this case we used an innovative application called GenieReader[1], which is introduced in details in Section 2.2. Given this scenario, we compared the selected frameworks (a justification for selection is given in Section 2.1) based on their various performances in solving complex tasks that (usually) require the collaboration of multiple agents. For this, we applied various metrics (cf. Section 2.5), for instance, latency and execution time. In addition, we also show a comparison to a well-known baseline setting: Vanilla RAG [1] (cf. Section 2.6). This is a specific realization of the general RAG approach, well-considered in the community.

Finally, the paper summarizes our experience while working with and testing the frameworks, providing hands-on take home messages for the use of MASs in real-world applications.

## 2 Experimental Settings

### 2.1 Selection of Frameworks

In the theoretical observations, we have shown an overview of different MAFs. All those frameworks focus on multi-agent collaboration and use different AI models, mainly LLMs. However,

[1] `https://genie-enterprise.com/geniereader`, last check: January 30, 2025.

they differ in how scalable they are, the resources they need, and their training capabilities. These differences make each framework better suited for specific tasks based on complexity and available resources.

For the experimental settings, we have chosen LangGraph [2], CrewAI [3], and SmolAgent [4]. All of these frameworks are open-source and free to use and provide protocols and methods for agents to exchange information effectively by imitating human-like interaction. In terms of workflow, the three representatives show prototypical characteristics of frameworks agent collaboration, or capabilities. Providing more details, we selected these particular frameworks for the following reasons.

- **LangGraph** [2] enables communication and reasoning of the graph-based agent. It supports conditional workflows with loops, where decisions dynamically influence the execution flow [5]. Agents are represented as nodes, with edges representing their interactions, making the framework well-suited for modeling complex, adaptive workflows.
- **CrewAI** [3] facilitates role-based agent collaboration by structuring agents into teams with well-defined roles. The key components of this framework include Agents (task performers), Tools (e.g., web scrapers, calculators), and crew (a team of agents), allowing a seamless execution of sequential and complex workflows [5].
- **SmolAgents** [4] is a lightweight framework designed to rapidly prototype agent interactions. Similarly to CrewAI, it enables a team-based environment with tool access for task execution. In addition, it has code-based reasoning, which allows agents to execute tasks by defining logic in Python code [4].

Using these frameworks, users can define agents with specific roles and coordinate them for collaborative work. Furthermore, these frameworks integrate well with the different AI models.

### 2.2 Introduction of Application

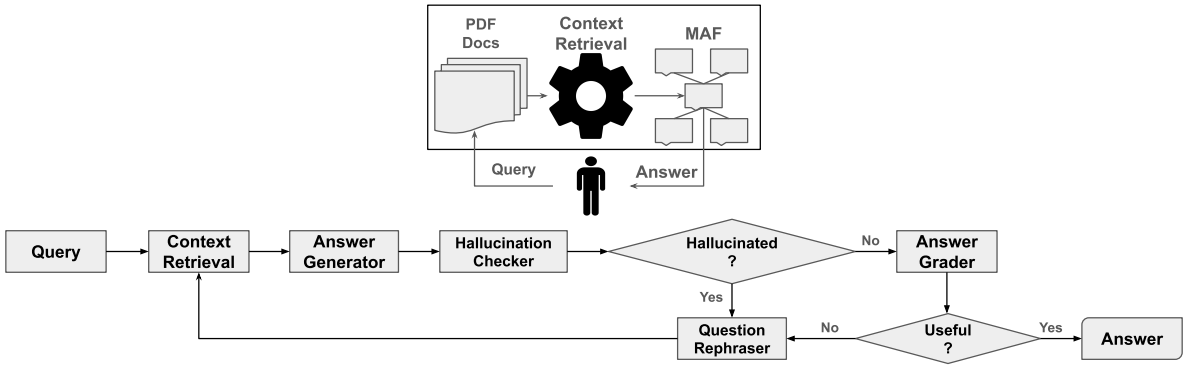
In this study, we analyze the effectiveness of various frameworks applied to an real-world, on market application, the GenieReader by Genie Enterprise, which is designed to address the challenges of managing and extracting information from complex documents.

Managing traditional paperwork or digitized datasets poses significant difficulties, particularly for documents exceeding multiple pages. Contracts, insurance documents, and other text-heavy materials often lack structured formats, making it challenging to extract precise information efficiently. Furthermore, documents containing tables and formulas present additional complexities, exacerbating inefficiencies in routine, paperwork-related tasks, consuming significant time and resources.

For this, GenieReader provides a comprehensive solution by leveraging artificial intelligence to support and automate routine tasks, including corporate compliance, due diligence, and contract management. The application enables efficient information extraction tailored to user-defined ontologies, ensuring relevance and accuracy. Additionally, GenieReader processes complex content, such as formulas and tables, by converting them into comprehensible formats. It also supports specialized research-oriented activities, including document mining, information research, and predicting case outcomes.

By integrating GenieReader into workflows, organizations can streamline document management, reduce resource-intensive tasks, and enhance accuracy in data processing. This technological innovation offers a competitive advantage, allowing users to embrace automation and focus on higher-value, strategic activities.

**Figure 1** – An overview of the agentic workflow adapted for this research. **Top:** A high-level representation of the adapted GenieReader workflow with MAF. **Bottom:** The generalized workflow applied to all MAFs in this study.




## 2.3 Agentic Workflow

In the GenieReader application, we provide an AI-powered question answering system. To achieve a better coverage of agent responsibilities, we decided to use four agents. The agentic workflow as shown in Figure 1 is a process that repeatedly improves responses by reducing errors (like hallucination, misinformation, etc.) and thus, enhances accuracy. To address a user query, the application employs a context retriever module (cf. Figure 1), which extracts relevant information from documents. The retrieved context will be passed through several agents: answer generator, hallucination checker, answer grader, and, if needed, question rephraser. Each agents finally ensures that the generated output is not only factually true but also contextually compatible [6].

The main four agents have their own functionality and below we briefly discuss them:

- **Answer Generator:** The agent generates an answer based on the retrieved context. Given the user query and relevant context, it tries to generate a response which should be informative, contextually compatible, and aligned with the evidence facts mentioned in the retrieved context.
- **Hallucination Checker:** The generated response should be grounded in the retrieved context, without containing any generated information [7]. We perform this check by cross-verifying the response with the source material and ensuring that it does not contain inconsistencies or hallucinated claims. If some part of the response is not supported, we discard that part and consider remaining parts are forwarded to the Answer Grader.
- **Answer Grader:** It determines the quality of the generated answer based on factors like factuality, completeness, coherence, etc. Here, we define certain grading rules to judge whether a given produced answer passes or fails. If a generated answer passes, it means the system's final answer is correct according to our expected behavior. In contrast, if a generated answer fails these checks then we will rank them using their confidence score probabilities (i.e., more confident answers will be ranked higher, believing they may have high probability of correctness.)
- **Question Rephraser:** This agent helps in reiterating the original user question if it fails to generate an answer. We recap the user question only if the retrieved context is not sufficient or it contains hallucinated information. This makes sure that the refined query retrieves better context for a next iteration of answer generation.

This process makes iterative refinements of the generated responses and reduces errors to increase trust in the given response [8]. Our system ensures that evidence-aligned and user

**Table 1** – Python package used and DSCS across different frameworks to communicate with Llama that was hosted on the Replicate server.

| Package | Frameworks | DSCS |
|---|---|---|
| LangChain | Vanilla RAG<br>LangGraph<br>CrewAI | 3,000 |
| LiteLLM | SmolAgents | 512 |

expectation-aligned responses are provided by techniques such as answer validation and query review.

### 2.4 Model Selection and Hyperparameters

All experiments were conducted using the Llama-3.1-405B-Instruct (Llama) model [9], which was hosted on the Replicate[2] platform. This model handled various tasks, including answering generation and agent-specific subtasks within the workflow (cf. Section 2.3). In all frameworks, the model temperature was set to 0.0. A key hyperparameter that varied was the Domain-Specific Context Size (DSCS) (cf. Table 1), which determined the number of tokens retrieved from the context retriever module of GenieReader (cf. Section 2.2) that could be passed to agents for answer generation.

In order to communicate with Llama using Replicate, we used LiteLLM[3] package for SmolAgents instead of LangChain[4] because of compatibility issues (cf. Table 1). SmolAgents also had an embedded system prompt that defined the logic behind the task execution in the form of Python code. The prompt was very long and contained multiple examples that helped the LLM with the reasoning process. The utilized version of Llama had a context window of 4,096 tokens. The number of tokens for the SmolAgents was set at 512 after multiple trials, while others were set at 3,000 tokens to provide appropriate response generation.

### 2.5 Introduction of Experimental Indicators

As mentioned previously, we evaluated the performance of different MAFs within the GenieReader application. To assess their effectiveness, we utilized the following key metrics:

- **Latency:** Measures the delay in communication or response time between agents during task execution, expressed in seconds.
- **Hallucination:** Assesses the extent to which an agent generates incorrect, irrelevant, or unsupported information. It factors reliability and factual accuracy on a 0–1 scale, where 0 means no agreement with the hypothesis and 1 means full agreement. This was computed using the implementation of [10].
- **Execution Time:** Captures the total time required to complete a workflow, measured in seconds.
- **Resource Utilization:** Represents a composite metric that quantifies CPU, memory, and network usage on a scale 0 to 1, where 0 indicates idle usage and 1 represents full system load. It is calculated as a weighted product of individual resource metrics as:

$$\text{Resource Utilization} = (0.4 \times \text{CPU}) + (0.4 \times \text{Memory}) + (0.2 \times \text{Network Activity}),$$

[2] `https://replicate.com/meta/meta-llama-3.1-405b-instruct`, last check: January 30, 2025.
[3] `https://github.com/BerriAI/litellm`, last check: January 30, 2025.
[4] `https://github.com/langchain-ai/langchain`, last check: January 30, 2025.

where

$$\text{Network Activity} = \frac{\text{Bytes Sent} + \text{Bytes Received}}{\text{Max Network Speed} \times \text{Execution Time}}$$

Here, Bytes Sent and Bytes Received represent total data transferred during the workflow, Max Network Speed is 780 Mbit/s, and the execution time is the task duration.

- **Average Attempts:** Denotes the average number of attempts made to generate a response to a given question.
- **Semantic Similarity:** Evaluates how closely the generated output matches the expected answer. It is measured on a scale of 0 to 1, where 0 indicates no similarity and 1 means identical content, computed using the implementation of [11].

### 2.6 Introduction of Baseline: Vanilla RAG

Vanilla RAG [1] combines pre-trained or built-in knowledge of a language model with facts retrieved from external sources (like documents, Wikipedia, etc.). This mix helps LLMs to know the current or specific information that is unknown or not being learned during model training. Finally, it generates accurate answers using both the model's understanding and real-world information.

The advantages of Vanilla RAG are improved factual accuracy, better interpretability by using retrieved documents in generation, and ability to update knowledge independently of model updates. In terms of response quality, RAG can generate more specific and diverse responses than pure parametric models (pre-trained model memory). Disadvantages include an increase in computational overhead due to retrieval, reliance on external data quality, and retrieval errors that potentially affected generation quality. Some existing pipeline systems also have strong generative decoders but weak retrievers, so they may not yet fully exploit the power of language models with retrieval [8].

In the baseline setting, only the LangChain framework was used for answer generation. The selection of the model and hyperparameters (cf. Table 1) remained consistent with those of the other frameworks. However, unlike the workflows in MAF, this setup did not include additional validation checks for the generated answers.

### 2.7 Data Description

All experiments were conducted with confidential data provided by an insurance client. The dataset consists of 28 insurance contracts in PDF format, written in German, containing highly specialized insurance terminology. Each document was accompanied by a questionnaire and its expected answers, provided in a separate Excel file, which had been annotated and verified by the client. These expected answers served as the ground truth for evaluating the semantic similarity of the generated responses. The PDF documents were uploaded to the GenieReader application, where relevant content for the user query was extracted as context using the context retrieval module.

### 2.8 Computational Settings

The experiments were conducted on a 64-bit Ubuntu system featuring an Intel i7 processor, 31 GB of DDR4 RAM, and an NVIDIA GeForce RTX 2070 GPU with 8 GB of VRAM, utilizing CUDA 12.1. The GenieReader application was executed by compiling Docker images built for specific tasks such as document ingestion, context retrieval, and question answering. Three

**Table 2** – Comparison of results using Vanilla RAG and different frameworks. In the table following abbreviations are used: HS … Hallucination Score, ET … Execution Time, RU … Resource Utilization, SS … Semantic Similarity, AA … Average Attempts.

| Implementation Type | Latency | HS | ET | RU | SS | AA |
|---|---|---|---|---|---|---|
| Vanilla RAG | - | **0.291** | **15.107** | 0.0584 | **0.7676** | - |
| LangGraph | 4.062 | 0.265 | 22.877 | 0.0559 | 0.6823 | 1.72 |
| CrewAI | 4.066 | 0.257 | 22.652 | 0.055 | 0.7501 | 1.65 |
| SmolAgents | - | 0.086 | 108.65 | **0.0034** | 0.6777 | **1.3** |

separate branches were forked from the original Vanilla RAG setup, each incorporating the selected frameworks for this research.

All experiments were conducted in a chat-based manner, using the questionnaire from the provided dataset. For each application response, the question, the retrieved context, and the generated answers were logged into a separate file, along with system metrics such as CPU usage, memory usage, bytes sent and received, latency, and overall execution time. These logs were later used for evaluation purposes.

## 3 Experimental Results

Table 2 presents a comparison of different frameworks tested with the GenieReader application. Latency was not computed for SmolAgents, as the manager in this framework handled all tasks nearly half the time instead of delegating them to team members. Similarly, Vanilla RAG relied solely on an answering agent. As a result, these two frameworks were evaluated on the basis of execution time.

Latency represents the average execution time per agent (in seconds), while execution time reflects the overall time taken per framework to generate answers. Although LangGraph exhibited slightly better latency (cf. Table 2), certain questionnaires required multiple iterations of the graphical workflow loop before arriving at a final answer, which could not be obtained in the first short. In contrast, agents in CrewAI collaborated seamlessly to handle complex queries, requiring fewer iterations to generate responses. As a result, CrewAI achieved a slightly better overall execution time compared to LangGraph (cf. Table 2). SmolAgents, however, had significantly higher response times due to the extended processing time of LiteLLM when communicating with Replicate, which was slower than LangChain (cf. Tables 1 and 2).

SmolAgents consumed the least system resources compared to the other frameworks (cf. Table 2). Although it had the highest number of bytes sent (160.93 KB) due to its predefined, lengthy in-built system prompts, the lower volume of in-domain contexts resulted in the lowest average number of bytes received for answer generation (1238.52 KB). As a light-weighted framework, SmolAgents also had the lowest overall CPU usage (0.835%), whereas Vanilla RAG recorded the highest CPU usage (14.445%).

In contrast, the other frameworks could use longer in-domain contexts, but unlike SmolAgents, they did not include additional in-built system prompts. As a result, the total number of bytes sent was significantly lower: Vanilla RAG (81.13 KB), LangGraph (107.25 KB), and CrewAI (155.76 KB). However, the use of longer in-domain contexts led to more detailed and comprehensive responses, increasing the number of bytes received: CrewAI (6574.12 KB), LangGraph (4756.34 KB), and Vanilla RAG (4185.45 KB). This, in turn, resulted in higher resource consumption compared to SmolAgents (cf. Table 2).

The hallucination score was computed by evaluating the generated answer in relation to the context from which it was derived, using the implementation from [10]. At the time of writing, SmolAgents, when using LiteLLM with the Replicate API, had a limitation on DSCS. One factor contributing to this limitation was the built-in prompts in SmolAgents, which con-

sumed a large number of tokens. Insufficient context in the domain resulted in answers that lack clarity (cf. Table 2). While the other frameworks also did not achieve a significantly high overall score, some responses exhibited partial agreement between the context and the generated answer, reflected individual scores greater than 0.5.

Achieving agreement in the hypothesis is challenging due to the significant difference in token lengths between the context and the generated answer. However, the primary focus is on the semantic similarity between the generated and expected answers. SmolAgents faced limitations in the number of tokens that could be passed as in-domain context (cf. Table 1), making it insufficient for generating convincing responses in the insurance domain, particularly when tested with a real Question and Answer application like GenieReader.

Meanwhile, LangGraph struggled to generate responses for certain complex queries, often failing even after multiple iterations of its graphical workflow. As a result, both LangGraph and SmolAgents recorded lower semantic similarity scores. In contrast, CrewAI was able to generate responses to these complex queries in fewer attempts compared to LangGraph, enabling it to achieve a semantic similarity score on par with Vanilla RAG (cf. Table 2).

## 4 Take Home Messages and Conclusion

The application-based assessment focused on comparing Vanilla RAG with MAFs. In particular, we considered the use case of answering user queries specific to the insurance domain in German, applying the GenieReader application. While both RAG and MAF eliminate the need to fine-tune LLM for domain- and language-specific tasks, *context length* plays a crucial role in determining the quality of the generated answers (cf. Tables 1 and 2).

Although SmolAgents is light-weight and presents a promising approach with its code based reasoning, its *limitations regarding DSCS* when using LiteLLM with a Replicate-hosted Llama (at the time of writing) hinder its ability to generate convincing answers for insurance-related queries in German.

LangGraph required constructing a graph to define a workflow, whereas CrewAI involved creating a team of agents and assigning tasks to them sequentially. In contrast, SmolAgents required defining workflows and task assignments through lengthy prompts while also defining a team environment comprising of a manager and its assistants. While LangGraph and CrewAI assigned specific tasks to designated agents in each iteration, SmolAgents followed this approach only half the time. Considering Agent Specific Protocols (ASP), LangGraph and CrewAI will be preferred over SmolAgents as a MAF for our application.

While LangGraph and CrewAI effectively followed ASP, *LangGraph struggled* with certain *complex queries*. Even after multiple attempts at re-phrasing and re-generation, it often failed to produce an output. In contrast, CrewAI successfully generated responses for all queries regardless of complexity and required fewer attempts per query compared to LangGraph. Further, its semantic similarity score was nearly on par with that of Vanilla RAG (cf. Table 2).

The current version of the GenieReader application utilizes Vanilla RAG, and the feedback has been positive so far. However, future iterations will incorporate MAF to introduce additional validation checks for generated answers. Based on its performance in terms of semantic similarity with the expected answers (cf. Table 2) and the need to accommodate longer DSCS (cf. Table 1), CrewAI emerges as the preferred choice over other frameworks.

## Acknowledgement

We acknowledge support by the "SEPE" project (funded by the German Federal Ministry of Education and Research; grant number: 16SV9291).

## References


[1] YU, W., H. ZHANG, X. PAN, P. CAO, K. MA, J. LI, H. WANG, and D. YU: *Chain-of-note: Enhancing robustness in retrieval-augmented language models*. In Y. AL-ONAIZAN, M. BANSAL, and Y.-N. CHEN (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 14672–14685. Association for Computational Linguistics, Miami, Florida, USA, 2024. doi:10.18653/v1/2024.emnlp-main.813. URL https://aclanthology.org/2024.emnlp-main.813/.

[2] LANGCHAIN AI: *Langgraph: Multi-agent framework for llms*. 2025. URL https://github.com/langchain-ai/langgraph. Accessed: 2025-01-24.

[3] CREWAI INC.: *Crewai: Multi-agent framework for llms*. 2025. URL https://github.com/crewAIInc/crewAI. Accessed: 2025-01-24.

[4] ROUCHER, A., A. V. DEL MORAL, T. WOLF, L. VON WERRA, and E. KAUNISMÄKI: *'smolagents': a smol library to build great agentic systems.* https://github.com/huggingface/smolagents, 2025.

[5] DUAN, Z. and J. WANG: *Exploration of llm multi-agent application implementation based on langgraph+ crewai*. *arXiv preprint arXiv:2411.18241*, 2024.

[6] SHUSTER, K., S. POFF, M. CHEN, D. KIELA, and J. WESTON: *Retrieval augmentation reduces hallucination in conversation*. In M.-F. MOENS, X. HUANG, L. SPECIA, and S. W.-T. YIH (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 3784–3803. Association for Computational Linguistics, Punta Cana, Dominican Republic, 2021. doi:10.18653/v1/2021.findings-emnlp.320. URL https://aclanthology.org/2021.findings-emnlp.320/.

[7] JI, Z., N. LEE, R. FRIESKE, T. YU, D. SU, Y. XU, E. ISHII, Y. J. BANG, A. MADOTTO, and P. FUNG: *Survey of hallucination in natural language generation*. *ACM Comput. Surv.*, 55(12), 2023. doi:10.1145/3571730. URL https://doi.org/10.1145/3571730.

[8] LEWIS, P., E. PEREZ, A. PIKTUS, F. PETRONI, V. KARPUKHIN, N. GOYAL, H. KÜTTLER, M. LEWIS, W.-T. YIH, T. ROCKTÄSCHEL ET AL.: *Retrieval-augmented generation for knowledge-intensive nlp tasks*. *Advances in Neural Information Processing Systems*, 33, pp. 9459–9474, 2020.

[9] DUBEY, A., A. JAUHRI, A. PANDEY, A. KADIAN, A. AL-DAHLE, A. LETMAN, A. MATHUR, A. SCHELTEN, A. YANG, A. FAN ET AL.: *The llama 3 herd of models*. *arXiv preprint arXiv:2407.21783*, 2024.

[10] BAO, F., M. LI, R. LUO, and O. MENDELEVITCH: *HHEM-2.1-Open*. 2024. doi:10.57967/hf/3240. URL https://huggingface.co/vectara/hallucination_evaluation_model.

[11] REIMERS, N. and I. GUREVYCH: *Making monolingual sentence embeddings multilingual using knowledge distillation*. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2020. URL https://arxiv.org/abs/2004.09813.