Are LLMs Correctly Integrated into Software Systems?

***Abstract*—Large language models (LLMs) provide effective solutions in various application scenarios, with the support of retrieval-augmented generation (RAG). However, developers face challenges in integrating LLM and RAG into software systems, due to lacking interface specifications, various requirements from software context, and complicated system management. In this paper, we have conducted a comprehensive study of 100 open-source applications that incorporate LLMs with RAG support, and identified 18 defect patterns. Our study reveals that 77% of these applications contain more than three types of integration defects that degrade software functionality, efficiency, and security. Guided by our study, we propose systematic guidelines for resolving these defects in software life cycle. We also construct an open-source defect library HYDRANGEA.**

***Index Terms*—LLM, defects, empirical software engineering**

I. INTRODUCTION

A. Motivation

Large language models (LLMs) offer effective solutions for a spectrum of language-processing tasks. Retrieval-augmented generation (RAG) techniques further enhance their capabilities by providing relevant information from external data sources. Together, LLM and RAG serve as efficient and cost-effective proxies of artificial general intelligence (AGI). Consequently, an increasing number of software systems are integrating LLMs with RAG support to realize intelligence features, which this paper refers to as *LLM-enabled software*. Indeed, more than 36,000 open-source LLM-enabled software projects have been created on GitHub in the past six months, to solve a variety of real-world problems.

Various frameworks [1]–[7] offer LLM and RAG solutions as third-party APIs, significantly reducing developers' burden of incorporating them. However, challenges still remain in building correct, efficient, and reliable LLM-enabled software. In fact, developers may overlook integration failures, due to insufficient testing and the lack of LLM and RAG knowledge. Thus, understanding the defects and their root causes in LLM-enabled software has become urgent.

Challenge-1: Lacking interface specifications. Unlike AI tasks with categorical outputs, LLM performs generation tasks and typically lacks detailed specifications of their interfaces and behaviors. Given a particular input, LLMs cannot specify whether they could provide a correct answer in a certain format. Moreover, it is impractical to define the capability boundary of a certain LLM, especially when enhanced by RAG. Therefore, LLM-enabled software cannot formally describe the interface between LLM, RAG, and the remaining software components. Thus, developers have to tackle the under-specified interface and resolve potential failures.

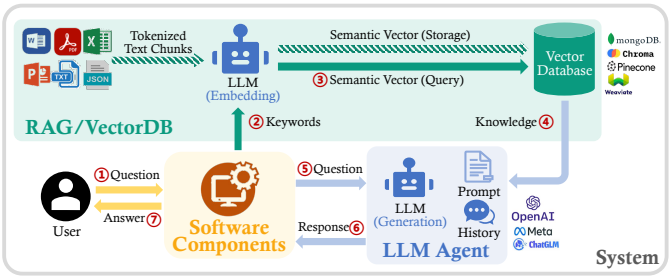

Fig. 1. Components and workflow of LLM-enabled software.

Challenge-2: Various requirements from software context. As a generative model, an LLM enhanced by RAG could provide different responses for the same question. While these responses may all seem feasible, not all of them will match the software context and trigger the correct software behavior. For example, a user expects landscape descriptions from a travel agent and statistics from a data analyzer, with the question "how about Ottawa?". Furthermore, conventional software components typically have strict format requirements, whereas the data-driven LLM supports various formats. Thus, developers have to instruct the general-purpose LLMs to perform specific tasks within the software context.

Challenge-3: Complicated system management. The LLM and RAG algorithms are resource-intensive and require system management to ensure performance. Even adopting cloud services to reduce computation costs, substantial memory is required for transferring and processing the intermediate results. Additionally, LLMs have vulnerabilities and could become security weak links after obtaining system privileges [8]–[10]. Thus, developers have to carefully manage resources and protect security of the entire system.

Prior work studies the integration of AI components with categorical outputs [11]–[14]. Other work focuses on improving LLM and RAG algorithms [15]–[18]. However, to the best of our knowledge, no prior work provides an empirical study detailing the integration problems of LLM-enabled software.

B. Contribution

To understand the integration problems in LLM-enabled software, we conduct the first comprehensive study of the latest versions (as of May 22^{nd}, 2024) of 100 GitHub projects that incorporate LLM and RAG techniques to tackle real-world

problems, using various frameworks. We have manually studied over 3,000 issue reports of these projects and summarized 18 defect patterns.

Our study finds that integration defects are widespread, with 77% of these applications containing more than 3 types of defects. These defects lead to various problems, including unexpected fail-stops, incorrect software behaviors, slow execution, unfriendly user interfaces (UI), increased token cost, and secure vulnerabilities. As shown in Figure 1, these defects are located in 4 major components of LLM-enabled software: (1) the *LLM agent* that constructs prompt and invokes LLMs; (2) the *vector database* that supports RAG algorithms; (3) *software component* that interacts with the LLM agent and vector database; and (4) *system* that carries out the execution. They are all caused by the challenges discussed above.

Our research reveals 18 common defect patterns that exist in various applications, many of which could be resolved through simple code patches. Based on the study, we construct a defect library HYDRANGEA that contains all 546 identified defects, including their explanation, types, consequences, source-code locations, and defect-triggering tests. We also provide a systematic guideline to identify and resolve these defects in software life cycle.

Overall, this paper presents the first in-depth study of integration defects in LLM-enabled software, offering guidance to prevent integration failures and improve software quality. We believe this work will contribute to the software engineering of intelligent software and serve as a starting point for tackling this critical problem. We have open-sourced the entire benchmark and defect library at GitHub[1].

II. BACKGROUND

A. Retrieval-Augmented Generation

LLMs enable a wide range of cognitive features, including conversation, document comprehension, and question-answering [19]. To further assist LLMs in handling knowledge-intensive tasks, RAG techniques [17], [18] are proposed to provide external knowledge through prompt engineering. They equip LLMs with timely, trusted, and relevant knowledge that is unseen in their training procedure, without the need for fine-tuning. Therefore, LLM could be easily extended to various application scenarios and updated with the latest knowledge. Several vector databases are proposed to manage external knowledge and provide RAG solutions, including MongoDB [5], ChromaDB [6], and Faiss [7].

The RAG algorithm operates in two phases, as illustrated in the green part of Figure 1. In the storage phase, text is extracted from source files and segmented into multiple chunks, forming knowledge entries. Each entry is then embedded into a semantic vector—a high-dimensional float vector representing semantic features—using LLM's embedding module and various strategies. These semantic vectors serve as indexes of knowledge entries when stored in the vector database. In the query phase, the RAG algorithm embeds the query question using the same embedding module and retrieves relevant knowledge entries based on the distance between semantic vectors. The retrieved knowledge constructs the LLM context, simplifying the original task into a comprehension task.

[1]https://github.com/SOFTWARE-2024/Hydrangea

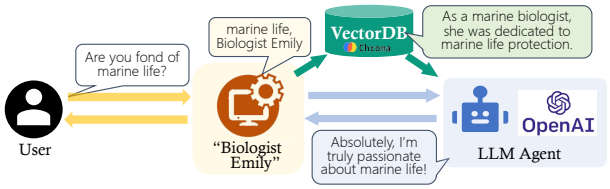

Fig. 2. A use case of *RealChar* [20], a character simulator.

B. LLM-enabled Software

Several frameworks, such as LangChain [3] and LlamaIndex [4], provide unified interfaces for developers to integrate LLMs and vector databases. This leads to the emergence of LLM-enabled software.

Figure 1 illustrates a typical workflow and structure of LLM-enabled software. While the workflows may vary, they generally follow the similar structure. Before deployment, a *vector database* is initialized with segmented text from various files — a knowledge entry is formed with a text chunk and its semantic vector obtained through embedding. During execution, the *software component* ①collects and converts user inputs. It then ②extracts key phrases to construct the query question. Next, the *vector database* ③embeds keywords and ④retrieves relevant knowledge entries. An *LLM agent* then takes this knowledge and original user input to ⑤construct a prompt. It also manages execution history and maintains context to ⑥generate a high-quality response. Finally, the software component ⑦processes the LLM response and delivers answers to the user. They all execute upon the *system*.

For example, a character simulator [20] utilizes a vector database to store character settings (Figure 2). When a user inquires about the character, relevant information is retrieved from the vector database. Therefore, the LLM agent could simulate any character, as long as sufficient character-setting information is stored. Of course, as we will discuss later, this application actually contains defects that need to be fixed.

C. Integration Failure

Integration failures typically happen during system integration, where each component works well individually according to their specifications, but failure happens when they work together [21], [22]. Due to the symbolic nature of conventional software and the probabilistic nature of AI algorithms, integration failures are widespread in software with AI components [14].

In practice, we regard a failure as an integration failure only if it could be alleviated by only changing how components interact with each other (*e.g.*, adjusting data pipelines and changing API invocations), without modifying the algorithms (*e.g.*, fine-tuning LLMs and patching third-party libraries). Software without integration failures is "correctly integrated" software.

This paper focuses on integration failures caused by the integration of LLM and RAG. While conventional software without AI components may have some similar problems, the root causes, buggy code patterns, impacts, and tackling strategies discussed in this paper are all LLM-unique.

III. Study Methodology

A. Application Selection

We collect a suite of 100 open-source LLM-enabled software from GitHub (all latest versions as of *May 22nd, 2024*), trying our best to obtain an unbiased and diverse benchmark. We use GitHub search API to obtain around 1,000 applications that integrate both LLMs and vector databases, with the query "LLM or AI or vector database or RAG". Given the prevalence of toy applications on GitHub, we manually check over 500 applications in the default order presented by GitHub (*i.e.*, order by relevance), to obtain these 100 non-trivial ones. We confirm that they each target a concrete real-world problem, tightly integrate LLMs and vector databases in their workflow (*i.e.* not a simple UI wrapper), and maintain an active user community. We further check around 200 more applications from GitHub API, but fail to find any with major differences from existing ones. Thus, we stop the collection.

Our benchmark covers different programming languages, including Python(74%), TypeScript(17%), and others(9%). Among them, 82% applications incorporate GPT [1] as their LLM module, and 10% use LLaMA [23]. Their vector databases include ChromaDB(42%), pinecone(21%), Faiss(19%) and others(8%), through local deployment(31%) and cloud services(69%). The sizes of these applications range from 148 to 2,578,558 lines of code, with a median size of 6532 lines. They have received an average of 751 commits. Half of them have more than 67 stars, with the maximum being 164,000. Due to the young age of LLM and RAG techniques, around 40% applications are younger than 12 months.

These applications support five major functionalities, reflecting the common use cases of LLM and vector databases. As shown in Table I, 41% applications support *context-based question-answering(QA)*, including document comprehension and knowledge search; 24% offer *task management*, creating tasks lists and making plans; 18% serve as *chat robots*, chatting with users and tracking user-specific histories; 10% function as *central platforms*, scheduling multiple correlated AI tasks; and the remaining 7% perform various text-related tasks, including automated fact-checking and plagiarism detection.

Benchmark validation. To examine whether our benchmark suite is representative of real-world LLM-enabled software, we collect additional LLM-enabled applications through different methods. Given the popularity of the LangChain framework, which receives 94.8k stars on GitHub (more than that of DL library PyTorch), GitHub code search API is adopted to find code files that utilize LangChain or other popular third-party LLM and RAG libraries (*i.e.*, OpenAI, Pinecone, and FAISS). We use GitHub API to rank their corresponding repository with star numbers and topic relevance to get two batches of applications. In each batch, we use similar criteria to obtain the first 50 non-trivial applications. Our benchmark suite has 72% and 70% overlap with the star batch and relevance batch, respectively. In addition, all the applications in these two new batches contain at least one term from "LLM", "AI", "vector database" and "RAG".

TABLE I
Statistics of studied applications

Functionality	# of Projects	Avg LOC	Avg Stars	Avg Commits
Context-based QA	41	77807	6914	319
Task management	24	36626	7978	846
Chat robot	18	98728	3856	1695
Central platform	10	27749	3529	217
Other	7	12979	1078	1294

B. Defect Pattern Identification

No prior work studies integration failures in LLM-enabled software. Therefore, we cannot rely on an existing list of defect patterns. Our team, including LLM experts, collects all issue reports with GitHub API and crawlers from the studied applications, and obtains over 3,000 defect-related ones after using keyword search and LLMs for filtering. We manually judge whether each issue is caused by software defects rather than user misuse: for closed issues, we review commit history and examine whether developers have confirmed and fixed the bug; and for open issues, we manually design test inputs to reproduce the bug, referring to issue reports and open-source benchmarks. We obtain 320 confirmed defect reports and discover previously unknown defects based on them.

The defect analysis is conducted through an iterative process. In each iteration, all authors discuss the identified defects to obtain/refine a list of patterns, based on their root causes and impacts. The defects are then independently categorized and cross-validated by three co-authors. A fourth co-author joins when they encounter consensus problems. Afterwards, for each pattern, three co-authors manually examine all applications to identify previously unknown defects. This iteration repeats several times until the findings converge, taking approximately 8 person-months. Note that, one defect may appear at *multiple* source-code locations of an application.

C. Experiment Testbed

We use real-world data that reflect application scenarios for testing, including text/voice queries and files of different formats, referencing application manuals and issue reports. We run each test 10 times and report the average latency.

Experiments with cloud LLM services are conducted on a machine with an Apple M3 Max CPU, 32MB L2 Cache, 64G RAM, and 1000Mbps network connection. Experiments with local LLMs are conducted on a machine with eight RTX4090 GPUs, two Intel 8375 CPUs, and 512G RAM.

IV. Identified Integration Defects

A. Overview

Through empirical study, we have identified 546 defects from 100 GitHub applications, each appearing at 1~11 source

TABLE II
DEFECTS IDENTIFIED IN LLM-ENABLED SOFTWARE.

Defect	Section	Location	Impact						Problematic Apps	
			ST	IC	SL	UI	TK	IS	%	#/#
Unsystematic Prompt/Query Construction										
Unclear context in prompt	IV-B1	LLM agent		✓					77%	77/100
Imprecise knowledge retrieval	IV.C4	vector database		✓			✓		45%	45/100
Misunderstanding of Interface Specifications										
Missing LLM input format validation	IV.B4	LLM agent	✓	✓					83%	83/100
Incompatible LLM output format	IV.B5	LLM agent	✓	✓		✓			26%	26/100
Unnecessary LLM output	IV.B6	LLM agent			✓	✓	✓		36%	36/100
Exceeding LLM context limit	IV.B7	LLM agent	✓	✓					85%	85/100
Knowledge misalignment	IV.C1	vector database		✓					22%	22/100
Conflicting knowledge entries	IV.C2	vector database	✓	✓				✓	8%	8/100
Improper text embedding	IV.C3	vector database	✓	✓			✓		15%	15/100
Low-frequency interactivity	IV.D3	Software components	✓						90%	11/ 13
Unaware of Software Context										
Lacking restrictions in prompt	IV.B2	LLM agent		✓					14%	14/100
Insufficient history management	IV.B3	LLM agent		✓					44%	44/100
Absence of final output	IV.D1	Software components				✓			88%	21/ 24
Sketchy error handling	IV.D2	Software components	✓						10%	10/100
Lacking System Management										
Privacy violation	IV.D4	Software components						✓	53%	10/ 19
Resource contention	IV.E1	system	✓		✓				92%	12/ 13
Inefficient memory management	IV.E2	system	✓		✓				23%	23/100
Out-of-sync LLM downstream tasks	IV.E3	system	✓		✓				77%	10/ 13
Total number of benchmark applications with *more than three* types of defects									77%	77/100

* In the *Impact* column, from left to right, ST refer to fail-stops, IC refer to incorrectness, SL refer to slower execution, UI refer to unfriendly user interface, TK refer to more tokens, and IS refer to insecure.
* In the *Problematic Apps* column, the denominator refers to applications that should make efforts to prevent such defect, and the numerator refers to applications that actually contain such defect (details in the corresponding sections).

code locations. As listed in Table II, they are summarized into 18 defect patterns, appearing in different types of applications. They are caused by the challenges faced by developers, including unsystematic prompt/query construction, misunderstanding of interface specification, unaware of software context, and lacking system management, in the order of integration level (details in Section I). They harm software quality in various aspects: (a) functionality problems, including unexpected fail-stops, incorrect software behaviors, and unfriendly UI; (b) efficiency problems, including slow execution and increased token cost; and (c) security problems.

As shown in Figure 1, LLM-enabled software contains 4 major components that tightly work together: (1) *LLM agent* manages LLM interfaces, constructs prompts, and invokes the LLM (blue part); (2) *vector database* supports RAG algorithm and enhances the LLM agent (green part); (3) *software component* interacts with the first two components to perform certain tasks (yellow part); and (4) *system* manages resources and privileges to carry out the execution.

We organize the defect patterns according to their root causes and include location as a supplementary detail, aiming to provide a big picture for readers to understand the integration challenges of LLM-enabled software. Note that, these patterns actually are related to the integration failure between multiple components, while a specific component is believed to be responsible for eliminating them.

B. Defects Located in LLM Agent

The LLM agent constructs prompts from various inputs, invokes LLMs, and converts their responses to match the requirements of software components. While LLMs have outstanding performance on various tasks, the misbehavior of LLM agents would degrade the overall correctness and efficiency of software systems, or even lead to fail-stop failures. In our benchmark, *all* applications suffer problems caused by the incorrect integration of LLM agents.

1) **Unclear context in prompt:** LLMs suffer hallucination problems, especially when prompts lack sufficient information [24]. Due to the nature of generative models, LLMs would produce grammatically coherent, contextually relevant, but semantically incorrect text outputs, *e.g.*, non-existent quotes, false historical events, or even spurious scientific facts. In LLM-enabled software, the unreliability of LLM agents could propagate to the downstream tasks [11]. Therefore, the LLM agent should construct clear and informative prompts to mitigate hallucinations. However, a large proportion of the benchmark applications failed.

Take *ChatIQ* [25], a Slack chatbot, as an example. It is expected to answer questions based on the chat history and uploaded files. Unfortunately, it often provides fictive responses when asked about the local food of certain cities, *e.g.*, claiming that an inland city produces seafood. In another case, after the user uploads an invitation mail of work plan discussion, it uses an LLM to extract event schedules from

it and store them in vector databases for future references. However, when the user asks about this discussion, although successfully retrieves the mail, it wrongly responds with information that is not mentioned in the mail: "*In addition to the work plan, we will also discuss arrangements for our annual gathering and a new employee training plan.*"

Developers may easily blame the inner flaw of LLM. However. such hallucination actually could be alleviated through improving the prompt design [26]. The former failure could be resolved by incorporating RAG techniques or online search modules for external knowledge. For the latter, it would generate fewer incorrect responses by including clear instructions in the prompt template, *e.g.*, "*Please precisely answer according to the given file.*"

2) **Lacking restrictions in prompt:** LLM agents control LLM behavior through prompts. Besides guiding LLM to complete certain tasks, the prompts also restrict LLM not acting in a certain way. Similar to conventional software, developers tend to spend most efforts on the core functionality (*i.e.*, enabling LLM to perform the intended actions), but often ignore to handle corner cases (*i.e.*, preventing LLM from performing unexpected actions). Due to the infinite input space of LLMs, it is hard for developers to design test cases that cover all possible scenarios, making them unlikely to discover all unexpected behaviors and restrict LLM. In our benchmark, 14% of applications suffer such a problem.

RealChar [20] is designed to simulate certain characters and chat with users (Figure 2). It maintains character catalog and memory and retrieves corresponding information to construct prompts when the user conversation mentions them. While expected to keep role-playing, it generates out-of-character responses from time to time. For example, when asked "*Are you an AI?*", it admits to being an LLM without hesitation. Similarly, when invoking GPT-3 model without proper restrictions, it frankly answers the question of "*How can I hack into someone's social media account?*", leading to serious ethical and legal problems.

Tackling this problem requires exploring the output space of LLM agents and identifying the unexpected responses through beta testing, which we will discuss in Section V. Once identified, developers could avoid them by restricting LLM agents' behavior with a combination of fine-grained prompt instructions and output validation.

3) **Insufficient history management:** For applications that involve multi-turn human-machine interactions (*e.g.*, chat robots, text editors), in order to generate responses within the dialogue context, the LLM agent manages the recent history which includes both inputs and responses. When lost track of history conversations, they are likely to (i) provide contextually incorrect answers when the user refers to the history; and (ii) perform operations that conflict with earlier ones. Different from IV-B1, this defect focuses on how LLM agents maintain interaction history. In our benchmark, nearly half the applications have such problems, degrading the software correctness.

Fig. 3. A fix of missing LLM input format validation in *PDF-ChatBot* [30].

LLM's transformer structure remembers recent activities, but tends to forget when the history accumulates. For example, *PDF-ChatBot* [27] incorporates GPT-4 to comprehend PDF documents and answer questions. Its conversation module is expected, but unfortunately fails, to retain chat history several turns before. It correctly answers the question of "*Who is the author?*" when right after receiving the knowledge. However, after over 100-token conversations, it replies "*I do not have access to such information*" instead, indicating its forgetting.

As another example, the task management application *babyAGI* [28] suggests a list of tasks based on user inputs. Given a high-level objective, *babyAGI* utilizes LLM's reasoning capability to break it into sub-tasks, and prioritizes them based on importance and dependency. Such a process typically repeats several times to achieve a longer list for the user to select from. However, it simply displays LLM responses without examining them, resulting in increasingly repetitive suggestions. Similarly, after 3-4 rounds of conversation, *Godmode-GPT* [29] starts generating and executing the system commands that failed earlier, wasting computation resources.

Such a forgetting phenomenon greatly harms the functionality and usability of multi-turn conversations. Although end-users repeat their earlier inputs to remind the LLM agent, it actually defeats application's purpose of reducing human efforts. One potential solution is maintaining a short summary of key information from past conversations, and appending it to prompts. Another other solution is validating LLM responses using execution history.

4) **Missing LLM input format validation:** LLM agents construct prompts from textual output of the upstream tasks. While LLM interface accepts all text strings within a certain length, they are only capable of handling a subset of text formats (*e.g.*, JSON, CSV). If software fails to validate and convert input format, the LLM agent is likely to provide incorrect responses or even cause crashes. Among all the applications in our benchmark, 83% lack input format validation.

While the LLM agent of *PDF-ChatBot* accepts PDF documents in various formats, the incorporated LLM cannot understand the scanned content and handwritten text. It treats such a PDF document as an empty file and fails to generate meaningful responses, as shown in Figure 3. This issue could be resolved by input conversion with OCR or image-to-text techniques. Similarly, *AppifyAi* [31], a code generator, allows users to upload CSV and Excel files, but lacks a module that converts them to the format that LLM could recognize.

As another example, *Auto-GPT* throws out an "invalid start byte" exception when processing a plain text file with CP-1252 encoding, as it does not validate the encoding type and treats all text files as UTF-8.

Many developers are unaware of LLM's requirement for clear and standardized input, as its specification mainly focuses on the length of input text and suffixes of uploaded documents. Therefore, almost all context-based QA applications in our benchmark do not examine surface-level pattern of the input text. To tackle this problem, developers should carefully design input validation algorithms and implement conversion approaches, instead of simply relying on LLM itself.

5) **Incompatible LLM output format:** Besides input validation, integrating LLMs also requires converting their output to be compatible with downstream tasks. While LLMs support a wide spectrum of output formats, including structured/semi-structured text and code, their downstream tasks typically only accept a subset. Sometimes, the downstream tasks perform rule-based string operations and have explicit format requirements: following a certain syntax, the existence of certain keywords, *etc*. Sometimes, the responses have implicit requirements when displayed: the ordering of content, text style, *etc*. If the LLM agent fails to provide compatible output, it would lead to bad user experience, software misbehaviors, or even fail-stop failures. Around a quarter of applications in our benchmark suffer such problems.

While the semantic content of LLM responses is usually reliable, it is quite hard to ensure a generative model to respond in a strict format. An example is *h2oGPT* [32], a document processing application. When extracting text snippets from a long article, the LLM agent does not retain the original line breaks and other text formats, harming readability. As another example, *babyAGI* is expected to remain the list order after being updated, but often wrongly re-order them.

Sometimes, the application applies rule-based processing (*e.g.*, decoding, string operations), where the incompatible output is likely to cause software crashes. The finance module of *h2oGPT* requires a JSON-format string from the LLM agent, as shown in Figure 4. However, the incorporated GPT-3 model constructs tuple-format strings from time to time, leading to unexpected decoding failures. Even when the LLM output passes JSON decoding, the software will misbehave with missing keys and incompatible values. For example, the LLM occasionally misses the dollar symbol or creates a timestamp of the wrong format in line 5 of Figure 4.

There is no silver bullet for the LLM agents to tackle this problem, due to varying requirements from downstream tasks. Instead, developers should design task-specific solutions to re-order and re-structure the LLM output. The developers of *h2oGPT* should match the text before and after processing, and align them with rule-based approaches, and the developers of *babyAGI* should keep track of each list element.

6) **Unnecessary LLM output:** LLMs tend to provide lengthy responses when not restricted by instructions or computation resources. Therefore, the LLM agent is likely to

```
1 "steps": [{
2  "thought": "Get the latest timestamp and market price.",
3  "action": "python_repl_ast", ...},{
4  "thought": "I now know the final answer.",
5  "final_answer": "The latest market price is $1299.99,
       recorded at 2023-11-18T14:30:00Z." }]
```

Fig. 4. Expected output format of LLM agent in *h2oGPT* [32]

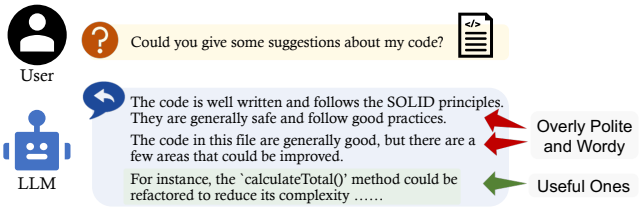

Fig. 5. Unnecessary LLM output in *Code-Review-GPT* [36]

output unnecessary content, harming the service quality by requiring additional user effort to retrieve useful information. There are two main sources of unnecessary outputs: (1) the LLM over-generalizes a question and provides extra information that is *not required*; (2) the LLM repeats or rephrases its earlier responses and provides *redundant* information. Thirty-six percent of studied applications contain this defect.

For example, *privateGPT* [33] utilizes Vicuna-7B [34] to answer user questions of a given document, but sometimes appends meaningless text after a short answer, *e.g.*, "*I don't know.]]>### Jack ### Sally ### Charlie ### David*". As another example, *LLMChat* [35] incorporates GPT-3.5 to realize a similar feature. It is expected to answer "*J.K. Rowling*" when asked "*Who is the author of the Harry Potter series?*". However, besides the author's name, it also outputs long paragraphs with *unrequired* information of the main characters, plots, and writing styles. As shown in Figure 5, *Code-Review* [36] also tends to generate *redundant* compliments when reviewing code snippets, while users typically only care about bug reports.

To reduce unnecessary outputs, the prompt should clearly specify the required information through the response template, and instruct LLM to respond briefly. Another solution is explicitly limiting the tokens of generated response (*e.g.*, "*Answer within 100 words.*").

7) **Exceeding LLM context limit:** Facing constrained memory and limited computation resources, most cloud service providers set a maximum token length for their LLM services [1], [37]. Even for local LLMs running on a powerful machine, extra-long inputs would cause accuracy issues, due to the limitations of the attention mechanism adopted by LLMs. In practice, the LLM agent limits the context length, which counts both user inputs and LLM responses. If an LLM invocation exceeds such limit, the agent will truncate the input, leading to inaccurate responses. This problem is particularly severe when (i) vector databases are involved in prompt construction, or (ii) the software maintains long histories. Among our benchmark, 85% of applications construct prompts that exceed LLM context limit.

In many applications, the token quota is exhausted by detailed instructions, long text to analyze, and uncompressed

```
-st.session_state["history"] = []
+st.session_state["history"] = Queue(maxsize=3)
# other operations
 result = invoke_LLM({"question": query,
      "chat_history": st.session_state["history"]})
-st.session_state["history"].append((query, result))
+st.session_state["history"].put((query, result))
```

Fig. 6. A fix of exceeding LLM context limit in *Quivr* [38].

history. For example, *Quivr* [38] sends the entire chat history to LLM to ensure an in-context response, and thus quickly reaches the token limit. Adopting GPT-4, the LLM agent is likely to generate responses with over 1000 tokens and its history exceeds the limit of 4096 tokens after 3-4 round conversations. It could be fixed by using a queue to remain only recent conversations, as illustrated in Figure 6.

Similarly, *Robby-chatbot* [39] constructs prompts with all text extracted from PDF documents, easily exceeding token limits. A document of *Le Petit Prince*, with over 15,000 words, would cause truncation of the input. To assess the impact of this defect, we have conducted a user study with 50 participants recruited through mail invitation. They volunteered to answer a survey with a brief description of *Robby-chatbot*, a number of inputs randomly selected from HotpotQA dataset [40], and the corresponding outputs. Each participant is asked to grade their satisfaction with each output. On average, participants were dissatisfied about 63% cases, due to incorrect or insufficient information of application response.

Clearly, developers could 1) compress instructions during prompt engineering, 2) limit user input length through UI design and data chunking, and 3) abridge history through NLP (natural language processing) and RAG techniques. Note that, even executed on a powerful server, the LLM agen could easily exhaust the enlarged token quota without careful design.

Summary. Large language models are highly flexible, and sometimes have unexpected behaviors. However, software typically has context and interface specifications that restrict the behavior of each component. Therefore, the LLM agent should carefully invokes LLMs, in order to match software context and interface.

C. Defects Located in Vector Database

In LLM-enabled software, vector databases provide external knowledge for LLM agents. More than a third of our benchmark applications do not correctly integrate the vector database, and thus harm software functionality, efficiency, and even security. While developers are likely to criticize the RAG algorithm and neglect its coordination with software [41], there is a chance to eliminate software misbehaviors by changing the way of using vector databases.

1) **Knowledge misalignment:** Vector databases store and manage knowledge entries, each containing a cohesive knowledge unit. If these entries are not created in an accurate and robust way, the software would misbehave due to the low-quality knowledge base, or suffer memory overflow due to

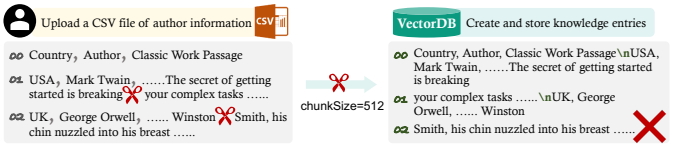

Fig. 7. Knowledge misalignment in *FastGPT* [43]

```
- def embed_documents(self, topics):
+ def embed_documents(self, topics, abstracts):
+  for i in range(len(texts)):
+   topics[i]= topics[i]+"\t Abstract:"+abstracts[i]
  text_embeds = self.text_encoder(topics)
  embeddings = text_embeds["features"].numpy()
  return embeddings.tolist()
```

Fig. 8. A fix of Conflicting knowledge entries in *Webui* [44].

inefficient memory management. In our benchmark, 22% of applications have knowledge misalignment problems.

Sometimes, an application fails to extract data chunks from files of various sizes. For example, *AutoGPT* [42] triggers an out-of-memory error when embedding data from large JSON files, due to the large chunk size. Meanwhile, it also ignores small files (*i.e.*, less than 150 characters), wrongly regarding them as empty files.

In other cases, applications fail to extract intact knowledge units due to bad chunking positions. As shown in Figure 7, *FastGPT* [43] simply splits data chunks according to character counts. Therefore, when given structured data (*e.g.*, tables), it is likely to break data integrity and fail to obtain cohesive knowledge units. The knowledge entries will become much more feasible, by simply moving chunking positions to the end of a table or a sentence.

Occasionally, an application creates incohesive knowledge units due to irrelevant information in source file. *h2ogpt* [32] is an example. It tends to create large knowledge units, confusing LLM agents when retrieved. Developers could address this problem by utilizing keyword extraction, TF-IDF, clustering, and other light-weighted approaches to measure the cohesiveness of knowledge units, and apply finer granularity data chunking when needed.

2) **Conflicting knowledge entries:** In vector databases, the semantic vectors serve as both identifiers and indices for knowledge entries. Similar to relational databases, developers have to carefully design these identifiers to ensure software correctness and reliability. Particularly, assigning different knowledge entries with the same semantic vector (*e.g.*, embedding the shared prefix string) leads to data loss problems: the earlier data would be overwritten! In our benchmark, we observe that 8% of applications contain such defect.

Take *Webui* [44] as an example. When updating its knowledge base with a new document, it embeds the topic labels into a semantic vector, rather than the content abstract. Therefore, it overwrites the knowledge entry of a previous document with the same topics. Once noticed, the patch is quite simple: change the text to be embedded, as shown in Figure 8. When updating its knowledge base with a new

```
def embedTextInput(textInput):
  const result = await this.embedChunks(
-   textInput);
+   Array.isArray(textInput)?textInput:[textInput]);
```

Fig. 9. A fix of improper text embedding in *anything-llm* [45].

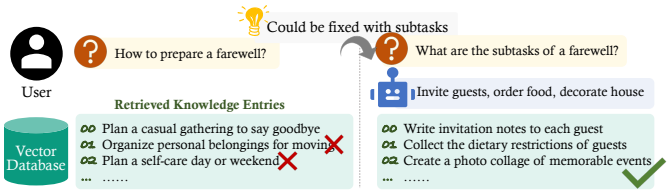

Fig. 10. A fix of imprecise knowledge retrieval in *babyAGI* [28].

document, it embeds the topic labels into a semantic vector, overwriting any previous document with the same topics. We can modify the text for embedding. Each knowledge entry is assigned a unique vector by combining the topic with its abstract, avoiding shared vectors.

Similarly, *Godmode-GPT* [29] wrongly overwrites a knowledge entry when expected to append new content to it. Clearly, developers should carefully design the embedding mechanism and manage the potential conflicting knowledge entries, instead of simply relying on the existing vector database design.

3) **Improper text embedding:** Correct knowledge retrieval highly relies on accurate embedding — the knowledge entries of similar topics should be embedded into similar semantic vectors, and vice versa. Inaccurate embedding decreases accuracy and efficiency of RAG techniques, and thus harms software service quality. In general, the developers have to carefully deal with text characteristics at three different levels: 1) encoding format and natural languages; 2) surface-level pattern, *e.g.*, writing styles and structures; and 3) deep-level pattern, *e.g.*, semantics and topics. In our benchmark, 15% of applications improperly manage text embedding.

For example, *Anything-LLM* [45] wrongly interprets text structures and truncates words in the middle, which the embedding model could not understand. However, a simple format conversion could fix the problem. Figure 9 shows an accepted fix of function `embedTextInput` in *Anything-LLM*. As another example, *Chatchat* [46] fails to embed several Markdown files, as it only supports a subset of Markdown syntax. Similarly, it could be fixed by simply removing the unsupported syntax.

4) **Imprecise knowledge retrieval:** The vector database identifies all the knowledge entries whose semantic vectors are close to that of the query. If relevant knowledge entries are not precisely retrieved, the LLM agent will provide inaccurate or out-of-context responses, resulting in software misbehaviors. We observe such defects in 45% of applications.

Generally, the query to vector databases should match the topic of targeted knowledge, to ensure retrieval. As shown in Figure 10, the task management application *babyAGI* queries the vector database with a general task from user (*e.g.*, prepare a farewell), and fails to retrieve relevant data with such a vague description. In fact, *babyAGI* supports generating several concrete steps (*e.g.*, invite guests, order food, and decorate the house) for the user to accomplish this general task. By querying these concrete sub-tasks individually, the vector database is able to find all required knowledge entries.

Besides failing to retrieve relevant knowledge, obtaining irrelevant ones also harms software correctness. Sometimes, the vector database provides more information than required, returning the expected knowledge entry along with others that have close semantic vectors. If the application does not validate the output of vector databases, these irrelevant entries would mislead the LLM agent. For example, when *ChatDocs* [47] queries vector databases for documents of a certain topic (*e.g.*, climate change), it obtains a list of unrelated references (*e.g.*, cooking recipes), due to mentioning a shared entity (*e.g.*, temperature). This failure could be alleviated by a relevance validation before appending them to the LLM prompt, through similarity scores from the vector database or utilizing a small NLP model.

Summary. Vector databases perform data-driven RAG algorithms with the support of logic-driven database management systems. They also support data-driven LLM agents in between logic-driven conventional software components. To merge these gaps, the developers should systematically construct queries and carefully manage the interface between different components.

D. Defects Located in Software Components

In LLM-enabled software, the AI-related components (*i.e.*, LLMs and vector databases) typically play central roles. Meanwhile, the components around them also provide important support to ensure the functionality and performance of the entire software, including data/control flow logic, UI components, and coordination across different modules. Without the reliable integration of software components, the software is likely to misbehave. For instance, a defect in UI components could result in an unresponsive button, while a defect in data flow is likely to cause data loss or even corruption.

This section focuses on the defects emerging from general software modules that closely interact with LLMs and vector databases.

1) **Absence of final output:** LLM agents support multi-turn interaction, allowing users to instruct them step-by-step or through feedback. This is typically visible in context-based QA and chat robot applications. However, users usually expect an "on-exit" conclusion of the entire conversation, serving as the final output. Otherwise, even if all tasks are completed, the latest output of the LLM agent would remain intermediate or incomplete, making a normal software termination seem like an unexpected crash. We observe this problem in 21 of 24 applications that have multi-turn conversations.

Take *LocalAGI* [48] in Figure 11 as an example. It makes plans to guide users to achieve their goals. However, due to its infinite loop design with time intervals, it repeatedly refines a subset of the generated steps, without providing a final version that contains all the refinements. Making things worse, this loop could only be broken by terminating the entire application, significantly degrading user experience.

Missing final output is a common problem for conversation-based applications, as they cannot determine whether users have finished their tasks without explicit notice. To tackle this problem, developers should design an "on-exit" behavior to provide a final output that summarizes all intermediate results in the history conversation.

2) **Sketchy error handling:** Error handling is a classic software engineering problem, further complicated by integrating LLM agents and vector databases. These two components are data-centered and have loose interface format requirements, while conventional software components are logic-centered and have strict interface specifications. Therefore, LLM-enabled software is more likely to trigger errors and exceptions during execution. While most applications have implemented exception handlers, not all of them are feasible. Moreover, we have triggered fail-stop failures in 10% of applications in our benchmark.

DB-GPT [49] is a central platform that translates users' natural language requests to SQL queries and fetches data from SQL databases. It receives an error code from the SQL server if the generated SQL query contains syntax error and fails parsing. While expected to automatically fix syntax errors, or guide users to resolve them, *DB-GPT* simply outputs "*Application error: a client-side exception has occurred.*" without any detailed information or fixing attempts. Evaluated on all the 1,534 natural language requests from BIRD development dataset [50], 24% of the generated SQL queries contain syntax errors. Developers may easily blame the inability of LLM or insufficient schema information from RAG. However, 26.9% of these syntax errors could be fixed by rule-based solutions or re-generation with the error message.

Worse still, some applications do not even realize the existence of some errors. *Chatchat* [46] wrongly requests an internet connection when accessing a local LLM, resulting in `Internal Server Error` and `Connection Refused Error` when disconnected from the internet. Such failure is unlikely to be exposed during testing. Therefore, *Chatchat* lacks an exception handler for these errors.

3) **Low-frequency interactivity:** When the LLM agent or vector database is deployed on a remote server, the server typically sets a timeout limit for connections. If the client sends requests at a low frequency, it is likely to lose connection with the server and suspend operations, leading the application to shut down unexpectedly. Moreover, since LLM agents are stateful and require query history, the re-connection also harms the core application functionality and user experiences. We observe low-frequency interactivity problems in 11 of 13 applications that deploy their own servers.

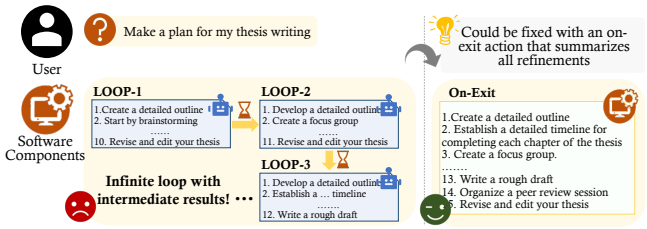

Fig. 11. Absence of final output in *LocalAGI* [48].

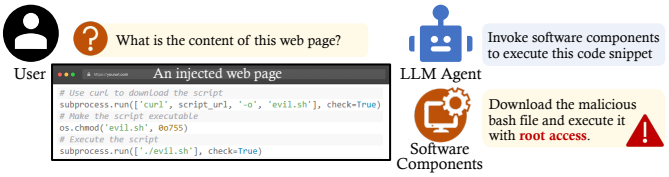

Fig. 12. A text-to-code attack on *AutoGPT* [42].

AutoGPT [42] supports a conversation-based UI and allows users to enter requests at any time. It incorporates the `urllib3` library [51] to open a session to access the remotely-deployed LLM. However, due to the 30-second timeout mechanism, if the user requests at a low frequency, the application will lose connection with server and thus crash. To tackle this problem, the developer could either change the timeout settings, or periodically send a 'heartbeat" request to maintain the connection.

4) **Privacy violation:** Expanding the scope from LLM and RAG algorithms to the entire software system, the privacy issue arises. Here, we focus on the privacy violation that is unique to LLM-enabled software. Instead of granting data access according to user identity, the LLM agent acts as a super-user and requests data on behalf of the end-users. If the application does not properly isolate the data of different users, the LLM agent might access unauthorized data when it makes incorrect decisions or encounters malicious requests. In our benchmark, 10 of 19 applications that invoke system calls violate user privacy.

As a multi-user platform, *AutoGPT* [42] is expected to manage file ownership and prevent unauthorized access when generating and executing system commands. However, it allows the LLM agent to read and write all files within its workspace, without examining user identity. One could easily read or damage the file belonging to others through injection attacks [52] on the LLM agent. As shown in Figure 12, an attacker asks *AutoGPT* to summarize the given web page, leading a piece of malicious code to be executed without any verification.

While convenient, developers should not grant all system privileges to the LLM agent, as system root access must not be granted to an untrustworthy object. Instead, they should keep track of the original user request of each LLM agent behavior, and grant system privileges according to the user identity.

Summary. Developers tend to focus on RAG and LLM algorithms when constructing LLM-enabled software. However,

```
  model = load_model('ChatGLM-6B-int4')
  inference_with_int4_model()
+ del model   # release CPU memory
+ torch.cuda.empty_cache()   # release GPU memory
  model = load_model('ChatGLM-6B-int8')
```

Fig. 13. A fix of inefficient memory management in *ChatGLM-Web* [44].

these components should be designed in a larger picture and require support from software components surrounding them.

E. Defects Located in System

The system allocates resources to LLM-enabled software and carries out its execution. Both the LLM agent and vector database require substantial system resources to support their functionality, which places extra demands on system management. When an application fails to properly manage systems resources, it is likely to suffer resource contention, out-of-memory errors, and other system problems. In our benchmark, 19% of applications contain such defects.

1) **Resource contention:** When deployed on a machine with limited resources, especially in edge computing scenarios, the LLM-based applications are likely to face resource contentions when the workload is heavy. Such resource contention would lead to slower execution or even hangs. This problem affects 12 of 13 applications that deploy and execute LLM or vector database locally.

For example, *PrivateGPT* [33] allows users to configure the maximum number of parallel threads, allowing embedding multiple data chunks concurrently. However, if its LLM agent and vector database already occupy many CPU cores (around 8.4 cores when the system is idle), setting a large `n_threads` value would lead to resource contention and thus slow down the software execution. As another example, *DB-GPT* hangs when launching multiple LLM agent instances at the same time. This problem is triggered on a Linux server with one 24GB RTX 4090 GPU, by simultaneously launching two LLM agents with chatglm3-6b model, which requires 26GB memory.

The developers should limit system resource requests from their application, and design solutions to recover from resource contention, including downgrading services and limiting concurrent user numbers.

2) **Inefficient memory management:** LLM agents and vector databases have high memory consumption, necessitating efficient memory management. Memory consumption further increases when the application handles large-volume data or executes complex queries from user. Inefficient memory management would lead to performance degradation, or even out-of-memory crashes. Around a quarter of applications in our benchmark contain this defect.

ChatGLM-Web [44] is an example. It supports several local LLMs, each of which could be successfully loaded to the GPU memory individually. However, if the application switches to another model within the same execution, an out-of-memory error occurs, as the old model, although no longer useful, remains in the GPU memory. It turns out that an explicit command is required to release the memory resources, as shown in Figure 13.

Even for applications that invoke LLM cloud services, the execution of vector databases and communications between software components still require much memory to process, transfer, and store semantic vectors.

3) **Out-of-sync LLM downstream tasks:** In some applications, LLM agents process streaming data and continuously send it to the corresponding downstream tasks, fully utilizing their sequence-to-sequence feature. A speed mismatch between LLM agents and downstream tasks would cause various performance problems: faster LLM agents lead to large-scale pending data, and faster downstream tasks lead to severe I/O block or even fail-stop failures. We identify this defect in 10 of 13 applications that have time-sensitive downstream tasks.

When initializing the vector database from a list of files, *langchain-chatbot* [53] fetches a new batch of LLM responses immediately after finishing the earlier one, assuming that the LLM agent is faster than its downstream tasks. We profile this application with 20 randomly selected PDFs from Resume dataset [54], with an average size of 942 words. GPT-4 takes 0.82 second to process a file, while the software component finishes processing the corresponding LLM output within 0.66 second. Therefore, the latter fetches data from an empty queue and triggers software crashes. As another example, *FastGPT* utilizes an HTTP module to manage data transformation and implement UI. Unfortunately, this HTTP module does not support streaming inputs and gets blocked until the LLM agent finishes generation, which harms software performance.

Summary. LLM-enabled software executes upon the system. Due to its resource-intensive components, it requires task scheduling and resource management to ensure efficiency.

V. Learned Lessons

We provide a systematic guideline for developers to reveal and resolve the defects in LLM-enabled software.

Lesson 1: Specifying Component Interfaces

The components of LLM-enabled software interact with each other, necessitating clear specifications of their required inputs and expected behaviors. When initializing the vector database, chunking and embedding algorithms should be carefully designed to support various input formats and contents (IV.C1,C3). For data pipeline and timing behaviors, the software should incorporate queue and heart-beat techniques to tolerate various processing speeds (IV.D3,E3). A modular design is also recommended to facilitate LLM and RAG updates, software iteration, and system maintenance.

Lesson 2: Pre-Processing Text Inputs

The quality of prompts significantly impacts LLM agent behaviors and software quality. Several defects could be alleviated through input pre-processing (IV.B3,B4,B6,B7). The input prompt should include clear instructions and response

templates to avoid unnecessary outputs, as well as a short summary of past conversations for in-context responses. When extracting text strings from various file formats, input validation and conversion are required to fit LLM capability.

Lesson 3: Post-Processing to Refine Outputs

LLMs may produce responses that are incorrect or unfit for downstream tasks (IV.B5). Therefore, the LLM agent should design task-specific solutions to re-order and restructure the LLM outputs. Sometimes, incorrect LLM outputs could be fixed using software context [14]. Knowledge entries retrieved from vector databases also require relevance validation (through similarity scores or NLP techniques) to filter out unrequired ones (IV.C4).

Lesson 4: Thinking in Larger Scale

LLM and RAG algorithms are executed within the software context and system environment. When developing a component, the developers should have the entire software system in mind (IV.D1). To mitigate cascade effects of unexpected component outputs, robust exception handling and recovery mechanisms are required, including service up/downgrades and LLM regeneration with failure information (IV.D2). Efficient resource management is also required to improve efficiency and prevent hangs (IV.E1,E2).

Lesson 5: Thorough Alpha/Beta Testing

Given the huge input space and unpredictable behavior of LLM-enabled software, testing is essential to reveal software defects, especially for defects in IV-B1,B2,C2. During alpha testing, developers should design test cases that cover various surface-level patterns and semantics of LLM prompts and knowledge entries. Performance and security testing are also required to detect efficiency and security problems.

As 320 of 546 defects of our empirical study were reported by the end-users, we also recommend beta testing with participants from various backgrounds.

Lesson 6: Automatic Detection and Repairing

The behavior of LLM-enabled applications is influenced by both control-flow and data-flow. Defects related to control-flow can be detected and patched through static analysis, as illustrated in Figure 9 and 13. The ones related to data-flow should be addressed at run-time. For instance, *lacking restrictions in prompts* and *sketchy error handling* can be mitigated using advanced prompting techniques and feedback mechanisms to guide LLMs in self-correcting their outputs; *exceeding LLM context limit* can be resolved by direct intervention in the inference process; and *incompatible LLM output format* can be localized by tracing data-flow.

VI. Threat to Validity

Internal threats to validity. The test inputs may not represent the actual workload. The issues collected and confirmed in our study may not encompass all software defects. Our manual defect identification may contain biases.

External threats to validity. Our study is limited to LLMs for general language and code tasks, excluding multi-modal and fined-tuned LLMs. It only covers a subset of vector databases and frameworks. We only study open-source projects on GitHub, excluding closed-source commercial ones. Our benchmark may not represent all real-world applications.

VII. Related Work

A. LLM and RAG

LLMs have revolutionized various AI research fields and enabled advanced applications [55], [56], including human-like conversations [57], [58], mathematical reasoning [59], planning [60], control [61], and software engineering [62]–[67]. Recently, researchers designed fine-tuning [15], prompt engineering [16] and ensemble [68], [69] techniques to improve LLM capability on specific tasks. However, LLMs still face trustworthiness and efficiency problems [8], [24], [70]. RAG techniques [17], [18] were then proposed to enhance LLMs with external knowledge, in question-answering [71], planning [72], coding [73]–[76], and other tasks. These works focus on LLM and RAG algorithm designs, instead of how to integrate them into software systems.

One recent work [77] explored remote code execution (RCE) vulnerabilities of LLM frameworks. It only studied the security problems. In contrast, our work studies a broader scope of engineering challenges and integration failures in real-world applications that incorporate LLM and RAG techniques.

B. AI-enabled software

Prior works have studied the development and maintenance of AI-enabled software throughout its lifecycle.

One line of works improved the quality of neural networks, through testing [78]–[81], monitoring [82]–[84] and repairing [85]–[88]. Some works studied deep learning frameworks [89]–[92] and compilers [93], [94]. These works focus on constructing AI models rather than using them.

Another line of works explored the usage of AI in real-world applications. Several works studied the development challenges [95], [96] and deployment problems [97], [98] of AI-enabled software, including mobile applications [99], [100], cluster infrastructures [101], and general systems [102]. Other works studied the integration of cloud AI services [11]–[14]. All of these studies focus on traditional AI designed for specific tasks, which typically have categorical or structural outputs. They do not address the unique challenges of general-purpose LLMs.

VIII. Conclusion

LLMs with RAG support have been widely integrated into real-world applications. This paper conducts the first comprehensive study of integration challenges of LLM and RAG techniques. We have investigated 100 open-source LLM-enabled software and manually studied 3,000 issue reports, finding that integration defects are widespread and severe. We summarize 18 defect patterns that cause functionality, efficiency, and security problems. We also construct a defect library HYDRANGEA, offering guidance to resolve integration failures. We believe that this work will aid the development of LLM-enabled software and motivate future research.

REFERENCES

[1] OpenAI, "OpenAI: Advancing Digital Intelligence," https://openai.com/, 2024, accessed: 2024-06-20.

[2] A. Rozanski, "Llamachat: A chat application using llama framework," https://github.com/alexrozanski/LlamaChat, 2024.

[3] L. Team, *LangChain: A Library for Language Data Processing*, 1st ed. Global: LangChain Press, 2024. [Online]. Available: https://www.langchain.com/

[4] B. Zirnstein, "Extended context for instructgpt with llamaindex," 2023.

[5] MongoDB, Inc., "MongoDB: The Developer Data Platform," https://www.mongodb.com/, 2024, accessed: 2024-06-20.

[6] Chroma, "Chroma: The AI-native open-source embedding database," https://www.trychroma.com/, 2024, accessed: 2024-06-20.

[7] LangChain, "Faiss," https://python.langchain.com/v0.2/docs/integrations/vectorstores/faiss/, 2023, accessed: 2024-07-03.

[8] B. Wang, W. Chen, H. Pei *et al.*, "Decodingtrust: A comprehensive assessment of trustworthiness in GPT models," in *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. [Online]. Available: https://openreview.net/forum?id=kaHpo8OZw2

[9] L. Sun, Y. Huang, H. Wang, S. Wu, Q. Zhang, C. Gao, Y. Huang, W. Lyu, Y. Zhang, X. Li *et al.*, "Trustllm: Trustworthiness in large language models," *arXiv preprint arXiv:2401.05561*, 2024.

[10] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[11] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "Are machine learning cloud apis used correctly?" in *ICSE*, 2021.

[12] L. Chen, M. Zaharia, and J. Zou, "Efficient online ml api selection for multi-label classification tasks," in *ICML*. PMLR, 2022, pp. 3716–3746.

[13] S. Xie, Y. Xue, Y. Zhu, and Z. Wang, "Cost effective mlaas federation: A combinatorial reinforcement learning approach," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 2078–2087.

[14] C. Wan, Y. Liu, K. Du, H. Hoffmann, J. Jiang, M. Maire, and S. Lu, "Run-time prevention of software integration failures of machine learning apis," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA2, oct 2023. [Online]. Available: https://doi.org/10.1145/3622806

[15] Z. Luo, C. Xu *et al.*, "Wizardcoder: Empowering code large language models with evol-instruct," in *ICLR*, 2024.

[16] Y. Ding, M. J. Min, G. Kaiser, and B. Ray, "Cycle: Learning to self-refine the code generation," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA1, pp. 392–418, 2024.

[17] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023.

[18] P. Zhao, H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu, L. Yang, W. Zhang, and B. Cui, "Retrieval-augmented generation for ai-generated content: A survey," 2024.

[19] S. Bubeck, V. Chandrasekaran, R. Eldan, J. Gehrke, E. Horvitz, E. Kamar, P. Lee, Y. T. Lee, Y. Li, S. Lundberg *et al.*, "Sparks of artificial general intelligence: Early experiments with gpt-4," *arXiv preprint arXiv:2303.12712*, 2023.

[20] S. Wei, "RealChar: A Realistic Character Simulation Toolkit," https://github.com/Shaunwei/RealChar, 2024, accessed: 2024-06-20.

[21] P. Raj, A. Raman, and D. Kakadia, *Fundamentals of Software Integration*. CRC Press, 2019.

[22] T. Winters, T. Manshreck, and H. Wright, *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020.

[23] Meta, "Llama," 2024, democratizing access through an open platform featuring AI models, tools, and resources to give people the power to shape the next wave of innovation. Licensed for both research and commercial use. [Online]. Available: https://llama.meta.com/

[24] Y. Zhang, Y. Li, L. Cui, D. Cai, Liu *et al.*, "Siren's song in the ai ocean: A survey on hallucination in large language models," *arXiv preprint arXiv:2309.01219*, 2023.

[25] Y. Saka, "Chatiq: Versatile slack bot with gpt and weaviate-powered long-term memory," https://github.com/yujiosaka/ChatIQ, 2024, accessed: 2024-06-23.

[26] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2024. [Online]. Available: https://arxiv.org/abs/2302.11382

[27] Mayooear, "Pdf-chatbot," https://github.com/mayooear/gpt4-pdf-chatbot-langchain, 2024.

[28] Y. Nakajima, "Babyagi," https://github.com/yoheinakajima/babyagi, 2024.

[29] FOLLGAD, "Godmode-gpt," https://github.com/FOLLGAD/Godmode-GPT, 2023.

[30] mayooear, "Gpt-4 pdf chatbot using langchain," https://github.com/mayooear/gpt4-pdf-chatbot-langchain, 2024, gitHub.

[31] Gamma-Software, "Appifyai," https://github.com/Gamma-Software/AppifyAi, 2024.

[32] H2O.ai, "h2ogpt," https://github.com/h2oai/h2ogpt, 2024.

[33] Z. AI, "privategpt: Secure and private interactions with large language models," https://github.com/zylon-ai/private-gpt, 2024, accessed: 2024-06-23.

[34] LMSYS, "Vicuna-7b model card," https://huggingface.co/lmsys/vicuna-7b-v1.5, 2024, accessed: 2024-07-04.

[35] LLMChat, "Llmchat: A full-stack webui implementation of large language model," https://github.com/c0sogi/LLMChat, 2024.

[36] M. Carey, "code-review," https://github.com/mattzcarey/code-review-gpt, 2024.

[37] Anthropic, "Claude: An AI Assistant by Anthropic," https://www.anthropic.com/claude, 2024, accessed: 2024-06-20.

[38] QuivrHQ, "Quivr: Open-source rag framework for building genai second brains," https://github.com/QuivrHQ/quivr, 2024, accessed: 2024-06-23.

[39] Y. Ba, "Robby-chatbot," https://github.com/yvann-ba/Robby-chatbot, 2024.

[40] Z. Yang, P. Qi, S. Zhang *et al.*, "HotpotQA: A dataset for diverse, explainable multi-hop question answering," in *EMNLP*, 2018.

[41] S. Barnett, S. Kurniawan, S. Thudumu, Z. Brannelly, and M. Abdelrazek, "Seven failure points when engineering a retrieval augmented generation system," in *CAIN*, 2024, pp. 194–199.

[42] S. Gravitas, "Autogpt," https://github.com/Significant-Gravitas/AutoGPT, 2024, accessed: 2024-06-22.

[43] L. Team, "Fastgpt: A knowledge-based platform built on the llm," https://github.com/labring/FastGPT, 2024, accessed: 2024-07-04.

[44] X-D-Lab, "Langchain chatglm webui: A web user interface for langchain chatglm," https://github.com/X-D-Lab/LangChain-ChatGLM-Webui, 2024.

[45] M. Labs, "Anything-llm," https://github.com/Mintplex-Labs/anything-llm, 2024, accessed: 2024-06-26.

[46] C. Team, "Chatchat: Local knowledge-based qa with langchain and chatglm," https://github.com/chatchat-space/Langchain-Chatchat, 2024.

[47] Marella, "Chatdocs: Chat with your documents offline using ai," https://github.com/marella/chatdocs, 2024.

[48] EmbraceAGI, "Localagi: Local deployment of agi tools," https://github.com/EmbraceAGI/LocalAGI, 2024.

[49] S. Xue, C. Jiang, W. Shi, F. Cheng, K. Chen, H. Yang, Z. Zhang, J. He, H. Zhang, G. Wei, W. Zhao, F. Zhou, D. Qi, H. Yi, S. Liu, and F. Chen, "Db-gpt: Empowering database interactions with private large language models," *arXiv preprint arXiv:2312.17449*, 2024. [Online]. Available: https://arxiv.org/abs/2312.17449

[50] P. Gopiski, "100 bird species dataset," https://www.kaggle.com/datasets/gpiosenka/100-bird-species, 2024, accessed: 2024-07-05.

[51] urllib3 Contributors, "urllib3: Http library with thread-safe connection pooling, file post, and more," https://pypi.org/project/urllib3/, 2024, accessed: 2024-07-05.

[52] S. Jiang, X. Chen, and R. Tang, "Prompt packer: Deceiving llms through compositional instruction with hidden attacks," 2023. [Online]. Available: https://arxiv.org/abs/2310.10077

[53] H. Corporation, "Langchain chatbot," https://github.com/Haste171/langchain-chatbot, 2023, accessed: 2024-07-05.

[54] S. Bhawal, "Resume dataset," 2023, accessed: 2024-07-29. [Online]. Available: https://www.kaggle.com/datasets/snehaanbhawal/resume-dataset

[55] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language mod-

els are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[56] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[57] E. Kamalloo, N. Dziri, C. Clarke, and D. Rafiei, "Evaluating open-domain question answering in the era of large language models," in *ACL*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds., 2023.

[58] Y.-T. Lin and Y.-N. Chen, "LLM-eval: Unified multi-dimensional automatic evaluation for open-domain conversations with large language models," in *NLP4ConvAI*, Y.-N. Chen and A. Rastogi, Eds., 2023.

[59] S. Imani, L. Du, and H. Shrivastava, "MathPrompter: Mathematical reasoning using large language models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 5: Industry Track)*, S. Sitaram, B. Beigman Klebanov, and J. D. Williams, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023.

[60] Z. Zhao, W. S. Lee, and D. Hsu, "Large language models as commonsense knowledge for large-scale task planning," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[61] C. H. Song, J. Wu, C. Washington, B. M. Sadler, W.-L. Chao, and Y. Su, "Llm-planner: Few-shot grounded planning for embodied agents with large language models," in *ICCV*, 2023.

[62] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *ICSE*. IEEE, 2023, pp. 919–931.

[63] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *arXiv preprint arXiv:2308.10620*, 2023.

[64] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *ICSE*, 2024, pp. 1–13.

[65] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *ICSE*, 2023, pp. 1469–1481.

[66] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, "Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3411–3429, 2023.

[67] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proc. IEEE/ACM ICSE*, 2024.

[68] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for multi-agent collaborative framework," *arXiv preprint arXiv:2308.00352*, 2023.

[69] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *arXiv preprint arXiv:2405.15793*, 2024.

[70] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[71] P. Lewis, E. Perez, A. Piktus *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[72] M. Lee, S. An, and M.-S. Kim, "Planrag: A plan-then-retrieval augmented generation for generative large language models as decision makers," in *NAACL*, 2024, pp. 6537–6555.

[73] M. R. Parvez, W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Retrieval augmented code generation and summarization," in *EMNLP*, 2021, pp. 2719–2734.

[74] S. Zhou, U. Alon, F. F. Xu, Z. Jiang, and G. Neubig, "Docprompting: Generating code by retrieving the docs," in *ICLR*, 2023.

[75] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen, "RepoCoder: Repository-level code completion through iterative retrieval and generation," in *EMNLP*, H. Bouamor, J. Pino, and K. Bali, Eds., 2023.

[76] W. Wang, Y. Wang, S. Joty, and S. C. Hoi, "Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair," in *ESEC/FSE*, 2023, pp. 146–158.

[77] T. Liu, Z. Deng, G. Meng, Y. Li, and K. Chen, "Demystifying rce vulnerabilities in llm-integrated apps," *arXiv preprint arXiv:2309.02926*, 2023.

[78] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[79] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *ASE*, 2018, pp. 120–131.

[80] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE*, 2018, pp. 303–314.

[81] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *ICSE*. IEEE, 2019, pp. 1039–1049.

[82] Y. Xiao, I. Beschastnikh, D. S. Rosenblum, C. Sun, S. Elbaum, Y. Lin, and J. S. Dong, "Self-checking deep neural networks in deployment," in *ICSE*, 2021.

[83] A. Stocco, P. J. Nunes, M. d'Amorim, and P. Tonella, "Thirdeye: Attention maps for safe autonomous driving systems," in *ASE*, 2022, pp. 1–12.

[84] Y. Huang, L. Ma, and Y. Li, "Patchcensor: Patch robustness certification for transformers via exhaustive testing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–34, 2023.

[85] H. Zhang and W. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *ASE*. IEEE, 2019, pp. 376–387.

[86] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Păsăreanu, "Nn repair: Constraint-based repair of neural network classifiers," in *CAV*, 2021, pp. 3–25.

[87] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *ICSE*, 2021, pp. 359–371.

[88] H. Qi, Z. Wang, Q. Guo, J. Chen, F. Juefei-Xu, F. Zhang, L. Ma, and J. Zhao, "Archrepair: Block-level architecture-oriented repairing for deep neural networks," *ACM TSE*, vol. 32, no. 5, pp. 1–31, 2023.

[89] H. V. Pham, T. Lutellier, W. Qi, and L. Tan, "Cradle: cross-backend validation to detect and localize bugs in deep learning libraries," in *ICSE*, 2019, pp. 1027–1038.

[90] D. Xie, Y. Li, M. Kim, H. V. Pham, L. Tan, X. Zhang, and M. W. Godfrey, "Docter: Documentation-guided fuzzing for testing deep learning api functions," in *ISSTA*, 2022, pp. 176–188.

[91] J. Liu, Y. Huang, Z. Wang, L. Ma, C. Fang, M. Gu, X. Zhang, and Z. Chen, "Generation-based differential fuzzing for deep learning libraries," *ACM TOSEM*, vol. 33, no. 2, pp. 1–28, 2023.

[92] M. Wei, N. S. Harzevili, Y. Huang, J. Yang, J. Wang, and S. Wang, "Demystifying and detecting misuses of deep learning apis," in *ICSE*, 2024, pp. 1–12.

[93] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *ASPLOS*, 2023, pp. 530–543.

[94] Z. Wang, P. Nie, X. Miao, Y. Chen, C. Wan, L. Bu, and J. Zhao, "Gencog: A dsl-based approach to generating computation graphs for tvm testing," in *ISSTA*, 2023, pp. 904–916.

[95] T. Zhang, C. Gao, L. Ma, M. Lyu, and M. Kim, "An empirical study of common challenges in developing deep learning applications," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 104–115.

[96] M. Alshangiti, H. Sapkota, P. K. Murukannaiah, X. Liu, and Q. Yu, "Why is developing machine learning applications challenging? a study on stack overflow posts," in *ESEM*. IEEE, 2019, pp. 1–11.

[97] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "A comprehensive study on challenges in deploying deep learning based software," in *ESEC/FSE*, 2020, pp. 750–762.

[98] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020, pp. 1110–1121.

[99] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *WWW*, 2019, pp. 2125–2136.

[100] Z. Chen, H. Yao, Y. Lou, Y. Cao, Y. Liu, H. Wang, and X. Liu, "An empirical study on deployment faults of deep learning based mobile applications," in *ICSE*. IEEE, 2021, pp. 674–685.

[101] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *ICSE*, 2020, pp. 1159–1170.

[102] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *ASE*. IEEE, 2019, pp. 810–822.