Comfrey: Mitigating Integration Failures in LLM-enabled Software at Run-Time

Yuchen Shao[§¶], Yuheng Huang[†], Jiazhen Zou[§], Yuling Shi[◇], Long Yang[§¶], Lei Ma[†‡], Ting Su[§], Chengcheng Wan[§¶] *

[§] East China Normal University, China [¶] Shanghai Innovation Institute [◇] Shanghai Jiao Tong University, China
[†] The University of Tokyo, Japan [‡] University of Alberta, Canada
{ycshao, jzzou, longyang}@stu.ecnu.edu.cn, {tsu, ccwan}@sei.ecnu.edu.cn
yuhenghuang42@g.ecc.u-tokyo.ac.jp, yuling.shi@sjtu.edu.cn, ma.lei@acm.org

Abstract

Due to the unrestricted outputs of LLMs and strict requirements of software components, integration failures are widespread in software that incorporates LLM agents and retrieval-augmented generation (RAG). Even seemingly correct LLM/RAG responses can trigger software misbehaviors if they violate these requirements.

In this paper, we conduct an empirical study to understand integration failures in real-world LLM-enabled applications. Guided by this study, we present Comfrey [1], a runtime framework that adapts the LLM agent and RAG responses to meet software requirements, serving as a middle layer between AI and software components. It automatically detects and resolves potential integration failures through a three-stage workflow, ensuring component compatibility. Our evaluation with a variety of open-source applications demonstrates that Comfrey detects 75.1% and prevents 63.3% of potential integration failures with 8.4% overhead, significantly outperforming the baselines.

Keywords

LLM, software integration failure, run-time patching

ACM Reference Format:
Yuchen Shao[§¶], Yuheng Huang[†], Jiazhen Zou[§], Yuling Shi[◇], Long Yang[§¶], Lei Ma[†‡], Ting Su[§], Chengcheng Wan[§¶] . 2026. Comfrey: Mitigating Integration Failures in LLM-enabled Software at Run-Time. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3787847

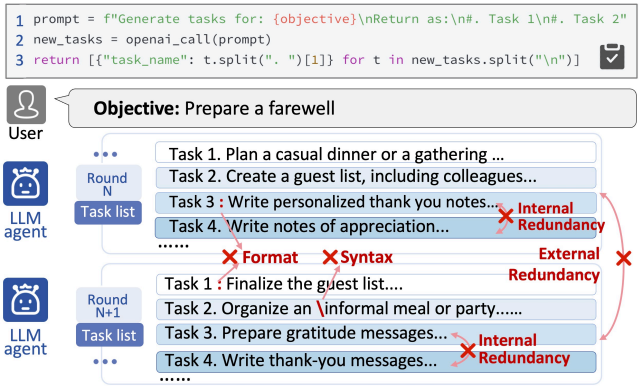

Figure 1: Task management application *babyagi* [6] encounters format/syntax/repetition errors.

1 Introduction

1.1 Motivation

Large language models (LLMs) offer effective solutions for processing and generating text, code, and other data. With the knowledge support from RAG (Retrieval-Augmented Generation), LLMs enable developers to build powerful tools for conversational agents [2–5], task management [6–8], program synthesis [9–11], and other applications. Therefore, they have been incorporated as the intelligent core in many AI software, referred to in this paper as *LLM-enabled software*. While effective, LLM-enabled software often faces *integration failures*, where the output of data-driven AI components (*i.e.*, LLM and RAG) is unaligned with the input specifications of logic-driven software components.

Integration failures are widespread in LLM-enabled software, as the LLM produces highly unrestricted outputs while its downstream software tasks often have strict requirements. The requirements cover three major dimensions [12–14]: *format*, *syntax*, and *repetition*. Format requirements come from the application scenario and data processing pipeline, which limit the valid styles of LLM/RAG output. For example, a voice assistant may expect its input to start with a certain phrase (*e.g.*, "Hi Siri").

The syntax requirement arises from general correctness restrictions on natural language and general programming languages; and the repetition requirements come from verbosity control and execution efficiency which directly affects user experiences. Moreover, LLMs are non-deterministic and could provide different outputs across invocations in terms of format, correctness, and coherence. This further challenges the robust integration of LLM.

To better understand these requirements, consider the task management application *babyagi* [6]. As shown in Figure 1, it constructs a prompt template (Line 1) that defines the output format, specifically requesting a numbered list of tasks. After invoking the LLM (Line 2), it extracts the task descriptions and strips the enumeration markers (Line 3) for display and storage. In the next iteration, *babyagi* refines these tasks to provide a more actionable plan.

*Chengcheng Wan is the corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3787847

This task generation pipeline seems simple, but it reveals the unalignment between the output of the AI component and the requirements of downstream tasks. In terms of *format* requirements, the parsing logic in line 3 strictly requires the tasks to start with "Task ", a number, a dot, and a space (*e.g.*, "Task 1. "). However, the LLMs wrongly use colons, commas, dashes, or other delimiters time by time, leading software to neglect the corresponding tasks. Moreover, LLMs may make *syntax* mistakes, such as unnecessary backslashes, which harm task interpretation. In addition, this case violates the *repetition* requirements in two aspects: (i) generates similar and hyper-focused tasks within a round, *e.g.*, "prepare gratitude messages" and "write thank-you messages"; and (ii) rewrites the task list with semantically similar alternatives, *e.g.*, "write personalized thank you notes" and "prepare gratitude messages".

While much work [15–18] has studied integration failures in conventional software systems, this problem remains unexplored for LLM-enabled applications. They do not tackle the integration challenges brought up by the non-deterministic behavior and complex data dependencies of LLMs and RAG. Several work [19–21] focuses on the API usage of AI with categorical outputs, but do not resolve the integration challenges of free-text outputs. Another line of work address the quality problems of LLM and RAG, including hallucinations [22, 23], incorrectness [24–27], inefficiency [28–30], context forgetting [31–33] and output inconsistency [34–36]. In parallel, a growing body of work explores prompt engineering [37–39] and instruction tuning[40–42] techniques to improve LLM responses. They only tackle the problems inside the AI component, but do not fundamentally resolve integration failures brought up by the software context. Recently, some work studies applying LLM in domain-specific tasks, including software engineering [43–46], conversational agents [47–49], scientific assistance [50–52], and clinical decision support [53–55]. However, they each target a specific task and cannot be applied to general LLM-enabled software.

1.2 Challenges

It would be beneficial to have an adapter that automatically aligns the output of AI components to the requirements of downstream components, preventing integration failures in LLM-enabled software. However, designing such an adapter has several challenges.

1. Various requirements from application scenarios. Given the wide spectrum of software implementation and application scenarios, the LLMs have to meet the dynamic and diverse requirements of software context. However, existing LLMs mainly provide domain-specific and task-oriented fine-tuned versions, which fundamentally lack the capability to address concrete software requirements, especially the format requirements. Therefore, a flexible middle layer is required to ensure compatibility.

2. Ambiguous software expectation of LLM/RAG output. It is inherently hard to precisely specify the behavior of the AI components [56]. Making things worse, their downstream software components only have ambiguous descriptions of their expected inputs. Therefore, it is hard to judge whether the LLM/RAG output aligns with software context.

3. Non-deterministic behavior of LLM agents. Due to the probabilistic nature of LLM and their agents, their accuracy descriptions are only statistically reliable. However, to ensure the correctness of the entire software, we have to determine whether a concrete output of the AI components would lead to expected software behavior. Deriving static code patches from LLM statistical information is insufficient in such scenario. Consequently, a run-time solution is required to tackle integration failures.

1.3 Contribution

In this paper, we first conduct an empirical study to understand the integration challenges of LLM-enabled software, as well as the symptoms of integration failures. Specifically, we analyze 50 open-source applications, each executed with 300 tests with human-labeled oracles. We find that 50.4% of the test inputs result in failures, and 74.8% of them are semantic ones that do not cause exceptions or other easy-to-observe symptoms. *All* failures are caused by improper integration: 52% are caused by violating format requirements, 14% syntax, and 34% repetition.

Guided by our study, we propose Comfrey, a run-time framework that prevents the integration failures in LLM-enabled software. Serving as a middle layer, Comfrey automatically detects and adapts the unaligned LLM and RAG outputs to meet software requirements.

The core algorithm of Comfrey follows a three-stage workflow where each stage tackles one type of error: format, syntax, and repetition. In each stage, Comfrey first obtains the corresponding requirement information, and then detects and repairs errors according to their symptoms. This workflow is modularized and could be easily extended to new software requirements. To tackle challenge-1&2, Comfrey captures the software requirements of AI components from software expectations and application scenarios, utilizing the specifications of surrounding software components to characterize the expected behavior of AI components. To tackle challenge-3, Comfrey always attempts to use non-AI solutions to tackle integration failures and adopts light-weighted AI-based solutions only when necessary, with the aim of minimizing the overhead and improving determinism.

We evaluate Comfrey on 100 open-source LLM-enabled applications that cover four domains, most of which incorporate RAG components. In our experiments, Comfrey achieves 75.1% recall and 96.6% precision in failure detection. After repair, the correct execution rate of the applications improves from 49.6% to 81.5%, achieving 19.0-64.0% more improvement than baselines with only 8.4% latency overhead.

2 Background

2.1 LLM Agent

LLM agents are autonomous components that employ LLMs to perform reasoning, planning, and task execution. To enhance capability, they dynamically invoke software components and access external knowledge sources when necessary [57]. Among the wide spectrum of LLM agents and their applications, there are four representative and popular types: conversational agents, task management, program synthesis, and context-based QA [58].

Conversational agents (*e.g.*, *chat-langchain* [59]) generate coherent and context-aware responses, enabling dialogue-like interactions with users. Task management agents (*e.g.*, *babyagi* [6]) help users to plan, prioritize, and execute goals by breaking high-level instructions into actionable steps and coordinating across other time

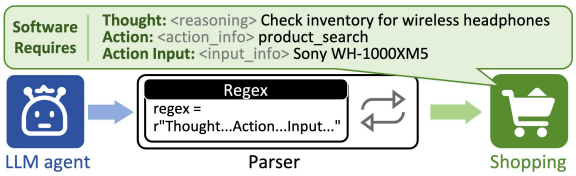

Figure 2: A shopping assistant application *h2ogpt* [62] and its requirement of LLM agent.

management tools. Program synthesis agents (*e.g.*, *LightGPT* [60]) enhance developer productivity by synthesizing code snippets or automating other tasks. Context-based QA systems (*e.g.*, *quivr* [61]) support high-precision information retrieval and response generation by dynamically retrieving domain-specific knowledge.

2.2 Retrieval-Augmented Generation

Given the limited internal knowledge of LLM, external knowledge is dynamically incorporated into prompts to enhance agent capabilities, enabling more accurate, contextually relevant, and evidence-grounded responses. Consequently, retrieval-augmented generation (RAG) is proposed to effectively store and retrieve semantically relevant knowledge content in a knowledge base, which is typically implemented as a vector database. Such knowledge base is usually constructed from a domain-specific corpus, such as academic literature, encyclopedia, and agent interaction histories.

The RAG pipeline has three major stages. During the *segmentation* stage, source documents are segmented into several chunks. To ensure RAG effectiveness, each chunk should contain exactly one knowledge unit, preventing semantic distortion. During the *indexing* stage, the chunk is stored in the vector database, indexed with its corresponding semantic vector obtained through an embedding model. During the *retrieval* stage, these indices are used for computing similarity with a given query from the LLM agent, in order to select the top-k relevant knowledge entries. It is expected to retrieve all required knowledge and filter out unrelated ones.

2.3 LLM-enabled Software

In LLM-enabled software, the LLM agent cooperates with other components to form a cohesive workflow, including data processors, decision algorithms, user interfaces, and other essential modules. Note that, we classify the LLM-enabled software with the same category as its core LLM agent.

Figure 2 illustrates a shopping assistant application *h2ogpt* [62], using a conversational agent to turn user input text into several structured action items. The downstream module then employs regular expression patterns to transform these items into machine-executable commands. Therefore, it strictly requires the output of LLM agent to follow the format of "Thought: <reasoning> Action: <action_info> Action input: <input_info>". Otherwise, it will simply neglect its input. Such error is also observable in other applications.

3 Empirical Study

3.1 Empirical Settings

3.1.1 Application selection. We study 50 Python applications that incorporate LLM agents to realize their core intelligence feature, covering the four major categories introduced in Section 2. We adopt their latest versions as of *May 28th, 2025*. Among these 50 applications, 33 are all the repositories with more than 50 stars in Hydrangea, a recently published benchmark of LLM-enabled software [63]. To create a balanced dataset, we additionally adopt 17 applications from GitHub that have the most stars in their categories through the same collection strategy and criteria of Hydrangea.

Table 1: Statistics of applications in empirical study

Type	# of Apps	% with RAG	Avg LoC	Avg Stars	Avg Commits
Conversational agent	12	75%	396,324	8,295	1,313
Task management	13	92%	92,689	16,502	1,358
Program synthesis	12	33%	23,629	851	91
Context-based QA	13	100%	99,862	17,856	1,471

As shown in Table 1, these applications are popular and actively maintained. On average, they have 130,885 lines of code, 899 commits in the recent 24 months, and 9,308 GitHub stars. Three-quarters of them integrate RAG algorithm to enhance their LLM agents.

For each application, we conduct testing on the entry function that invokes the LLM agent or RAG component. To ensure fairness of the comparison, all applications adopt Qwen2.5-32B model [64] for general tasks, Qwen2.5-Coder model for code-related tasks, and Qwen3-Embedding-0.6B model for RAG embedding.

3.1.2 Test input generation. We design 300 different test inputs for each application. All test inputs are selected and adjusted from existing AI datasets according to the application type. We use GPT-4o to judge whether each data entry matches the application scenario and software context, and adjust it if necessary. Two of the authors then manually review and refine all the tests. This creates 15,000 test cases across 50 applications, and takes 2 person-month effort.

1) Conversational agents. The conversation data are constructed from a multi-domain dialogue dataset MultiWOZ [65]. To expand its scenario, GPT-4o is adopted with a task-specific template designed by the authors (*e.g.*, "generate 10 variations of laws and business domain"). For each dialogue, only the user-side questions are retained as test inputs.

2) Task management. The task instances are constructed from Taskmaster-1 [66] dataset, which contains task-oriented dialogues. Since it only covers 6 scenarios, GPT-4o is adopted to synthesize 10 more scenarios, each with 300 dialogues, following the same style as the original one. Only the initial states and task goals are retained as test inputs.

3) Program synthesis. The code generation tasks are evenly selected from 3 code generation datasets: MBPP [67] for algorithmic reasoning, DS-1000 [68] for data science tasks, and APPS [69] for competition-level tasks. Only the natural language requirements are retained as test inputs.

4) Context-based QA. The context-question pairs are obtained from the NarrativeQA [70] dataset. To extend to real-world documents, we supplement it with BBC News [71] and arXiv [72], paired with reasoning questions generated by GPT-4o. The context is in the form of Word, CSV, and PDF.

We also tweak the tests for some special scenarios, including extremely short/long test inputs and mixed-programming-language inputs. For applications that support multi-turn conversations, we generate test inputs for 10 iterations.

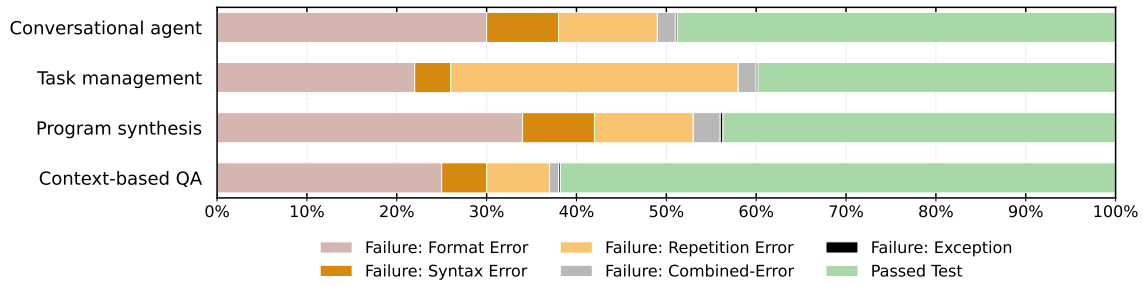

Figure 3: Testing results for 50 applications.

3.1.3 Failure Identification. We examine the data-flow and software decisions of each application and judge the correctness of each test. There are three key criteria: (1) data format compliance, (2) semantic correctness and coherence, and (3) balance between data simplicity and completeness.

We first leverage GPT-4o to examine the violations of downstream task requirements and user expectations, supplementing with related code and documentation [73]. We also adopt several heuristics for judgment, including regular-expression-based format checks and frequency analysis for repetition detection. Next, we manually verify the judgment result, particularly focusing on whether the overall software behavior aligns with the intended task logic and functional specification from the application documentation. Each test case and its corresponding oracle are cross-validated by at least two authors.

3.2 Testing Result Summary

Among 15,000 tests across 50 applications, 7,710 fail, with an average failure rate of 51% and a median of 60%, as shown in Figure 3.

All 50 applications encounter test failures. Task management applications have the highest failure rate of 60%, as they have constraints both within and across LLM agent responses. Program synthesis applications share a similar failure rate of 56%, as they require the code snippets to be enclosed in the response in a certain format. In contrast, context-based QA applications have the fewest failures (38%), as they typically have fewer requirements on the AI component outputs.

More than 99% of these test failures are semantic failures that do not cause exceptions or other easy-to-observe symptoms, like the example in Figure 1. This phenomenon further reflects the difficulty of detecting integration failures in LLM-enabled applications. This paper focuses on tackling these semantic failures, which have three major categories: format, syntax, and repetition. Among the 7,710 failed tests, 4% of them result from the combined effect of errors from different categories, which are denoted as *combined errors*. Despite the small proportion, these combined-error failures happen in half of the applications, covering all agent types, suggesting different errors may co-occur in real-world applications and thus require a systematic solution.

3.3 Type 1: Format Errors

Around half of the failures are caused by LLM/RAG outputs that fail to conform to the format requirements of the downstream software components. As these components typically have strict specifications of their inputs, the violations in format dimension would impede the parsing and processing of AI component outputs.

3.3.1 Template discrepancy. This is the most common type of format errors, contributing to about 30% of all failures. In order to parse the free text output from AI components, conventional software components typically expect LLM output to follow a certain template, in order to enable rule-based string processing (*e.g.*, regular-expression patterns).

In the example of Figure 2, *h2ogpt* requires the output of LLM agent to strictly follow the template of "Thought: <reasoning> Action: <action_info> Action input: <input_info>", which enables the regular-expression parsing in the downstream tasks. However, due to the non-deterministic nature of LLM, the LLM agent often generates responses not following the template, causing the downstream component to fail to parse such responses and wrongly terminate its invocation.

3.3.2 Improper data segmentation. Sometimes, an LLM-enabled application segments its input data into several fragments (*e.g.*, knowledge units and data entries) for analysis. These fragments are usually the output of the segmentation stage of RAG components. To maintain semantic coherence and integrity, such segmentation must align with natural linguistic or semantic boundaries. While such violation rarely triggers an exception, it degrades data-flow quality and hurts software functionality. This problem is particularly common in context-based QA applications, where improper segmentation accounts for roughly one-third of format failures.

LLMChatbot [74] exemplifies this problem through its extreme segmentation approach. It wrongly segments the text into character-level and treats each individual token as a knowledge entry. For instance, the word "quick" is separated into five individual characters. This prevents the LLM agent from obtaining any meaningful knowledge, and leads to incorrect responses to user questions.

3.3.3 Incorrect context construction. When the RAG component retrieves information from a large corpus, careful content filtering is required for constructing the context of LLM agents. There are two types of suboptimal constructions: exclusion of relevant content and inclusion of irrelevant/low-quality content. Besides interfering with LLM reasoning [75], it may also lead to context window overflow and truncated responses due to token accumulation. Around 40% of non-conforming output format failures are related to incorrect context construction.

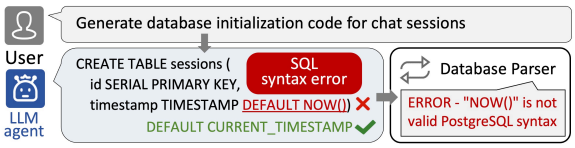

Figure 4: Syntax-parser misalignment in *LightGPT* [60].

Take a scientific document management application, *paper-qa* [76], as an example. It sets the relevance threshold to 0 when retrieving documents according to the user's question, which is likely to result in much irrelevant content.

3.4 Type 2: Syntax Errors

Unlike format errors, *syntax* errors are caused by violating the linguistic or grammar requirements of end-users and downstream tasks. Even strictly following a response template, LLMs could still deliver the same semantics with different syntactic constructions, some of which may not match the application scenario. Moreover, small syntax errors in LLM responses could cause downstream tasks to fail to parse their semantics, especially for compilers or tool-chain scenarios, leading to incorrect software behavior. Syntax errors contribute to 13% of test failures.

3.4.1 Syntax-parser misalignment. This type of error mainly happens in LLM agents that invoke compilers or other logic-driven tools. LLMs are designed for delivering semantics, instead of fulfilling the grammatical requirements, structural API specification, or other domain-specific, manually-defined rules. Therefore, a gap is likely to appear between the LLM-delivered semantics and the software-parsed information, *e.g.*, generate SQL queries with incorrect grammar [77]. We classify such errors as syntax-parser misalignment, which occupies 20% syntax errors.

Figure 4 shows an example from the program synthesis application *LightGPT*. When creating PostgreSQL database tables, it generates a SQL query containing syntax errors under default values, resulting in parser rejection.

3.4.2 Inconsistent lexical feature. Beyond software-imposed constraints, linguistic specifications introduce additional lexical requirements. Users typically expect consistent language standard (*e.g.*, varieties of English), language usage consistency (*e.g.*, without mixed languages), and conventions (*e.g.*, uniform paragraph/list structures) throughout a session. However, as LLM training corpus contains lexically diverse samples, LLMs cannot guarantee such consistency. This type of error accounts for 80% of all syntax errors.

The AI workflow automation platform *DB-GPT* [78] is a concrete example. It encounters inconsistent lexical features in 12% tests, including alternately using American and British English ("authorize" and "authorise") and responding in a different language from user query. Although these errors are of low frequency, they greatly hurt user experience once they appear.

3.5 Type 3: Repetition Errors

Repetition errors occur when LLM agents produce superfluous content beyond the required scope or rephrase existing output without providing extra information. These errors waste computational resources, increase latency, and confuse both end-users and downstream components. Repetition errors appear in nearly 30% of the failed tests. In task management applications, which frequently manage enumeration, they account for over half of test failures.

3.5.1 Redundant software behavior. Sometimes, the LLM agent repeats the same action to fetch or compute the content that is already known, or re-launches a tool with the same parameter. These redundant behaviors significantly reduce the software's efficiency. Note that we only record the actions and tools whose outputs are independent of the number of their invocations, *e.g.*, fetch the value of a constant. Around 15% of repetition errors belong to this category.

Take *AGiXT* [79] as an example. When a user asks for facts (*e.g.*, "the area of a certain country"), the application repeatedly invokes web search tools with the same keywords, nearly doubling the execution latency.

3.5.2 Redundant semantics. Repetition also happens in the dataflow, as illustrated in Figure 1. It accounts for 85% repetition failures. There are three forms: (1) internal redundancy, where a response contains duplicated sentences or semantically overlapping enumerations; (2) external redundancy, where new responses merely rephrase previous content without substantive changes; and (3) contextual redundancy, where a response contains verbose content irrelevant to the software context and application scenario [63]. They are particularly common when the software interacts with historical conservation data or manages lists.

The LLM agent in Figure 1 generates either semantically equivalent options or reordered duplicates when making plans. This redundancy not only distracts users during decision-making but also unnecessarily increases system execution time.

4 Comfrey Design

Based on Section 3, we propose Comfrey, a run-time framework that prevents the integration failures in LLM-enabled software.

4.1 Overview

As shown in Figure 5, Comfrey serves as the middle layer between the AI components (*i.e.*, LLM and RAG) and their downstream software components, automatically detecting and adapting the AI component outputs that violate the format, syntax, and repetition requirements. During runtime, Comfrey will be invoked at multiple code locations, with the workflow in Figure 5 being chainable across these invocations.

To integrate Comfrey, software developers only need to use its instrumentation API to automatically insert a Comfrey method after every invocation of AI component in the application source code. It also obtains the format, syntax and repetition requirements through static analysis. At run time, for every AI component output, Comfrey applies a series of requirement violation checking, utilizing the symptom patterns summarized in Section 3. Once a violation is detected, Comfrey converts such output to meet the requirements of downstream components, while preserving the semantics.

Stage-wise error tackling. The core algorithm of Comfrey follows a three-stage workflow, starting from resolving format errors (§4.3), followed by handling syntax errors (§4.4), and finally tackling repetition errors (§4.5). In each stage, Comfrey first obtains the

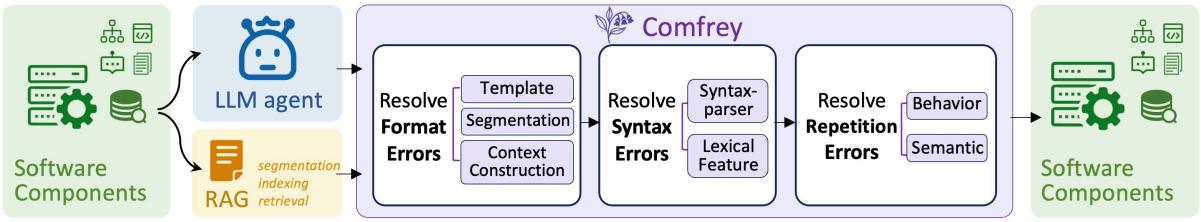

Figure 5: Overview of the run-time framework Comfrey.

corresponding requirement information. It then detects and repairs each type of errors one by one, as listed in Table 2. We adopt such a design to minimize the interference between error types, as well as tackle the failures caused by the combined errors. For error types that do not interfere with others, we prioritize the one that has more severe consequences.

Low-overhead design. As a run-time tool, Comfrey is designed to have low overhead. It always first attempts to use rule-based techniques that are computationally efficient, and avoids using computation-intensive techniques such as LLM self-regeneration and large-scale neural network inference.

It utilizes finite state automata and syntax tree analysis to detect format errors. For syntax errors, it incorporates compilers and parsers. It also incorporates an iteration-aware termination mechanism to early-exit the application loop that iteratively refines AI component outputs when repetition errors are detected. Comfrey utilizes computation-intensive techniques only when necessary. For example, it invokes a 0.6B-parameter embedding model only when it suspects the AI component output contains semantically similar content.

4.2 Obtaining the Requirements

The requirements of AI component outputs come from two sources: software expectations and application scenarios.

Extracting requirements from software expectations. While specification is an ideal source, most applications do not provide them. Instead, we extract requirements from the data processing code logic of downstream software components through static analysis. Comfrey first traverses the function call graph and conducts data flow analysis to identify all code locations where LLM/RAG outputs are consumed. Note that, it focuses on the branch edges that lead to the execution path of function's core functionality, ignoring the fall-through edges. It then conducts pattern-based static analysis on these code snippets to obtain the specific requirements for each error type: output template (§3.3.1), data chunk specifications (§3.3.2), context construction rules (§3.3.3), and parser syntax (§3.4.1).

Characterizing requirements from application scenario. There are also requirements from application scenarios, independent of the implementation. For example, enumerated outputs should avoid information duplication. Comfrey particularly focuses on 6 scenario requirements that arise from user expectations and application scenarios: intact textual elements(§3.3.2), and content relevance (§3.3.3) of *format* dimension; consistent lexical features (§3.4.2) of *syntax* dimension; and absence of unnecessary software behavior repetition (§3.5.1), succinct content (§3.5.2), and contextual semantic redundancy (§3.5.2) of *repetition* dimension. These requirements complement software expectations by ensuring outputs meet user experience standards and application-specific quality criteria.

4.3 Resolving Format Errors

4.3.1 *Template discrepancy.*

Requirements. The expected templates are obtained from the code snippet of downstream parsing. There are three major forms: (1) positional templates that have fixed identifiers and slots for filling (*e.g.*, Figure 2); and (2) structured-data templates that specify the formal schema (*e.g.*, JSON and XML); and (3) code-fenced templates (*e.g.*, Markdown code blocks).

Detection. Comfrey examines whether the AI output satisfies the expected template using finite state automata (FSA) [80]. If a violation is detected and the number of missing and extraneous elements (*i.e.* basic structural components) is smaller than the threshold $\tau_{element}$ (3 by default), Comfrey reports a format error. If there are multiple templates in the downstream task, it only examines the template with the smallest edit distance.

Repair. For positional templates, Comfrey first clusters the identifiers in AI output to those in the template, with respect to string-edit distances. It then re-orders the identifiers (together with the following slot) to fit the template requirement. For structured-data templates, Comfrey first refines the structure of AI output to match the template (*e.g.*, add a missing JSON key), using the minimum edits of element changes. It then applies type conversions to element whose content violate the schema. For code-fenced templates, Comfrey adds the missing delimiters and code block boundaries, as well as unifying the language identifier (*e.g.*, `python` and `py`).

Consider template in Figure 2 and the LLM output "Action:... Thought:...", Comfrey uses FSA to detect an invalid state transfer from "Action" to "Thought", and repairs by switching their order.

4.3.2 *Improper data segmentation.*

Requirements. Comfrey extracts the chunk size constraints and boundary markers (*i.e.*, sentence terminators and paragraph breaks) for data segmentation through static analysis on the processing operations in the earlier software components. There is also a scenario-driven requirement of integrity, where the segmentation should not break the sentence structure or separate a word into subwords.

Detection. This module is triggered only after the application performs text segmentation operations (*e.g.*, the segmentation stage of RAG). Comfrey validates whether all the requirements are fulfilled and reports an error when either of them is violated in any segment. As the chunk size and boundary marker requirements are naturally fulfilled, Comfrey focuses on the integrity. For each data segment, it

Table 2: Comfrey's strategies for detecting and preventing integration failures

Defect Type	Defect	Section	Detection Solution	Repair Solution
Format	Template discrepancy	§4.3.1	FSA validation with element threshold	Element re-ordering; structure refinement; delimiter supplementation
	Improper data segmentation	§4.3.2	Dictionary validation; syntactic tree analysis	Fragment bridging; sliding-window re-segmentation
	Incorrect context construction	§4.3.3	Two-stage similarity detection	Query-based relevance ranking; low-relevance entry removal
Syntax	Syntax-parser misalignment	§4.4.1	Compiler/parser syntax validation	AST refinement with minimal edit distance
	Inconsistent lexical features	§4.4.2	Language & structure examination	Translation; grammar correction; structure standardization
Repetition	Redundant software behavior	§4.5.1	History examination	Invocation bypass
	Redundant semantics	§4.5.2	Two-stage similarity detection	Content de-duplication; loop termination and rollback

refers to a dictionary when validating the integrity of the first and last words of the segment. It then constructs syntactic trees for the first and last sentences of the segment, and examines the missing linguistic elements, including dangling modifiers, incomplete noun phrases, and transitive verbs without subjects.

Repair. Comfrey tackles the integrity problem by copying the first/last word or sentence from adjacent segments and concatenating them with the fragment (*e.g.*, bridging "comput-" and "er"). During this process, the chunk size and boundary markers requirements might be violated. Once violated, Comfrey merges and re-splits adjacent segments with a sliding-window strategy, to make sure that each segment fulfills these requirements.

4.3.3 *Incorrect context construction.*

How to include all and only relevant information in the LLM context is one of the fundamental challenges in RAG systems. Although this complex problem extends beyond the scope of our work, Comfrey utilizes content relevance to identify some of the incorrect context construction.

Requirements. Comfrey assumes that all components in the LLM context should have relevant semantic (*e.g.*, sharing the same topics), which requires the RAG component to output data entries with high content relevance. This requirement comes from retrieval requirements of RAG and user expectation of coherent information.

Detection. Comfrey designs a two-stage similarity detection mechanism on the retrieved data of RAG component. It first computes the similarity score between every pair of data entries using TF-IDF similarity. If the score of a component pair is smaller than bottom quartile [81], Comfrey conducts second-round examination to further confirm the content relevance. Motivated by two recent work [82, 83], it computes the relevance score as the cosine similarity of sentence embeddings [84] of this pair. It reports an inclusion of irrelevant content if the similarity score is smaller than $\tau = 0.7$. This threshold is determined from empirical study to balance the precision and recall: lower values (*i.e.*, < 0.6) generate excessive false positives, while higher values (> 0.8) is likely to miss subtle redundancy patterns.

Repair. Comfrey tackles irrelevant content with the RAG re-ranking approaches [85]. When Comfrey identifies a pair of data entries with low relevance, it removes the one that is less relevant to the user query, using the similarity score.

4.4 Resolving Syntax Errors

4.4.1 *Syntax-parser misalignment.*

Requirements. The requirement arises from the parser that is applied to the output of AI components, including (1) compilers of a certain language; and (2) rule-based parser logic.

Detection. The detection of syntax-parser misalignment is straightforward. Comfrey proactively reuses the syntax checking module of the compiler/parser to validate AI component outputs before they undergo actual processing by compiler/parser. We focus on the syntax requirements in the branch edges for function's core functionality, ignoring the error-handling and fall-through edges. If there are multiple parsers/compilers in the downstream tasks, Comfrey reports an error if all of them are violated.

Repair. While repairing code syntax is a traditional software engineering problem, we focus on the errors that are common in LLM scenarios, as introduced in Section 3.4.1. Comfrey transforms AI outputs and syntax requirements into the AST domain for rule-based repairing. The high-level idea is to refine the AI outputs to match the syntax requirement that has the closest edit distance. Particularly, Comfrey tackles bracket and string literal mismatches, trailing commas, invalid operators, and incomplete expressions. Comfrey validates its repair attempt through re-compilation, and retries up to 3 times. Note that, Comfrey only repairs a subset of syntax errors that could be solved through automata.

4.4.2 *Inconsistent lexical features.*

Requirements. Comfrey focuses on the linguistic uniformity of three features throughout a session: (1) language usage, (2) language standard (*e.g.*, "authorize" vs. "authorise"), and (3) text structure. Comfrey establishes the standard using the lexical features of the first user text input, or the LLM prompt template when no user input exists.

Detection. For language usage, Comfrey utilizes Unicode script detection [86] to identify character ranges (*e.g.*, Latin, CJK, Arabic) and uses n-gram frequency analysis [87] against language models to detect language switches. Note that, we deactivate the language usage examination in translations, linguistic discussions, and other typical multi-lingual scenarios. For the language standard, it refers to a dictionary to examine whether the phrases belong to the same standard variety (*e.g.*, American/British English). For text structure, Comfrey examines the existence of subheadings, lists, and other structural elements. Comfrey reports an error when any of the standard is violated.

Repair. Once a violation is detected, Comfrey converts the violated part to meet standard. For language usage violations, Comfrey invokes a local translator. For language standard problem, Comfrey adopts the first repairing suggestion of a grammar checker [88]. For text structure, Comfrey follows the similar approach as Section 4.3.1 to align bullets, indentations and other basic structures.

4.5 Resolving Repetition Errors

4.5.1 *Redundant software behavior.*

Requirements. It is unnecessary to repeatedly invoke stateless and deterministic tools with the same input. Therefore, Comfrey maintains a list of tools and functions that are stateless and deterministic, relying on LLM internal knowledge and heuristics.

Detection. For each code location of LLM agent invocations, Comfrey maintains a history queue that records the last N tool/function invocations controlled by the agent, their parameters, and corresponding outputs (N is configurable and set to 10 by default). Once the agent invokes a tool for function, Comfrey searches the corresponding history queue for any past invocation of the same tool/function with the same parameter. Once detected, Comfrey reports a redundant software behavior if this tool/function is stateless and deterministic. Comfrey ignores those that may produce different results when the system state changes.

Repair. Comfrey bypasses the redundant tool/function invocation and uses the corresponding result in the history queue.

4.5.2 Redundant semantics.

Requirements. Comfrey assumes that the same semantic information should not be repeatedly delivered. It adopts the same similarity threshold $\tau = 0.7$ as §4.3.3, for semantic redundancy detection.

Detection. Comfrey follows the two-stage similarity detection mechanism of Section 4.3.3 to detect the redundancy. For redundant semantics, it examines three dimensions: (1) internal redundancy where two sentences in a response have a high similarity; (2) external redundancy where the responses of two iterations have a high similarity; and (3) contextual redundancy where the LLM prompt and response have a high similarity. Comfrey reports an error when any of them are detected.

Repair. For internal redundancy and contextual redundancy, Comfrey follows the same solution as Section 4.3.3. For external redundancy, Comfrey terminates the loop and rolls back to the value of the last iteration.

5 Implementation

We have implemented Comfrey for LangChain [89, 90] and Python applications, the most popular framework and programming language for developing LLM-enabled applications [56]. The core algorithm of Comfrey is general to various LLM/RAG algorithms and programming languages, including its static analysis, runtime monitoring and automated repair solutions. Comfrey can adapt to other programming languages by incorporating extra static analysis libraries (*e.g.*, ESLint [91] for JavaScript, SonarQube [92] for Java), and to different AI frameworks by implementing framework-specific API adapters.

Comfrey is a runtime Python library that utilizes function decorator [93] as its instrumentation API. To use Comfrey, developers only need to specify the code scope where Comfrey should take effect, without changing the software implementation. By default, the decorator should be applied to all the entry functions that invoke the LLM agent or RAG component. Comfrey then uses ByteCode module [94] and Python interpreter [95] infrastructure to insert detection and repair instructions between LLM/RAG output generation and downstream consumption.

Comfrey uses Pyan3 [96], Jedi [97], Python AST module [98], NetworkX [99] and Beniget [100] for static analysis. It uses `re` module [101] for regular expressions, NLTK [102] for tokenization and parsing, `ast`, and `dis` modules for syntax validation. It uses scikit-learn [103] and NumPy [104] for data analysis. It invokes Qwen3-Embedding-0.6B model through cloud API for semantic similarity analysis. The parameter settings of Comfrey (*e.g.*, thresholds and weights) follow the guidance of the original literature and are validated by the tests in empirical study.

All experiments are on a machine with an M3 Max CPU (4.05GHz, 16-core), 32MB L2 Cache, 64G RAM, a 2TB SSD, and 1000Mbps network connection. All LLM inferences are conducted through cloud APIs.

6 Evaluation

Our evaluation aims to answer the following research questions:
- *RQ1 (Detection):* How accurate is Comfrey's failure detection?
- *RQ2 (Repair):* How effective does Comfrey prevent failures?
- *RQ3 (Ablation):* How does each component contribute to Comfrey?

6.1 Methodology

6.1.1 Applications. We evaluate Comfrey on 100 open-source LLM-enabled applications, evenly distributed in application types. It includes all the 50 applications in our empirical study and 50 additional applications collected after the design of Comfrey. The additional sets are collected from GitHub, using the same methodology as Section 3.1.1. The two sets have the same average age (25 months), similar sizes (130,885 and 151,112 LoC) and popularity (9,308 and 10,831 stars). Section 6.5.1 compares the evaluation results of these two sets of applications.

6.1.2 Test data. We design a *new* set of 300 test inputs and their success/failure ground-truths for each application, following the same methodology in our empirical study and avoiding overlaps with the previous set in empirical study. The new test inputs lead to 15,124 failed tests across 100 applications, having similar breakdown as the previous set: around 51% of them caused by format errors, 12% by syntax errors, 33% by repetition errors, 4% by combined errors, and 38 are exception/crash-related.

6.1.3 Baselines. As there is no prior work tackling the integration failures in LLM-enabled software, we design one straw-man solution and adapt three recent techniques to our scenario.

(1) Retry: It retries LLM/RAG invocation with the same input if an exception occurs during the software processing the output.

(2) Reflexion [105]: It is a recent work of LLM self-correction, which asks LLM itself to judge the correctness of its output and repair the error. As it only focuses on the LLM component, we adapt it to the software scope by invoking it after each LLM invocation, providing the general text description of requirements (which is obtained by Comfrey) and error types.

(3) SmartGear [56]: It is a runtime tool that tackles the integration failures of traditional ML tasks. We adapt its repairing solutions for ML tasks with text output (*e.g.*, optical character recognition), which clusters the AI output to the focal values in path constraints, with respect to the edit distance.

(4) JSON Schema Prompting [106]: It is a preventive approach that provides structured output constraints to LLMs during generation by including JSON schema specifications in the prompt.

Table 3: Result summary across 30,000 testing runs.

	Detection				Prevention	
	TP	FP	Recall	Precision	Failures	Pass Rate
No Tool	-	-	-	-	15,124	49.6%
Comfrey	11,356	398	75.1%	96.6%	5,550	81.5%
Retry	38	0	0.3%	100.0%	15,086	49.7%
Reflexion	6,234	3,567	41.2%	63.6%	9,450	68.5%
SmartGear	3,245	1,678	21.5%	65.9%	11,234	62.6%
JSON Schema Prompting	4,446	1,934	29.4%	69.7%	10,678	64.4%

* *TP*: True Positives; *FP*: False Positives.

We adapt it by encoding general software requirements as unified JSON schemas, which are then embedded in prompts.

6.1.4 Repair judgment. Due to the huge cost of manual validation across all four schemas, we employ GPT-4o to evaluate repair correctness against ground-truth requirements [107]. The GPT-4o is provided with the ground-truth, precise software requirements, and detailed explanations of error types and corresponding examples. We derive confidence scores from the log probabilities, quantifying LLM's certainty for each judgment.

To ensure the reliability of our semi-automated labeling process, we implement a multi-stage quality assurance mechanism. For low-confidence LLM judgments (*i.e.*, $confidence < 0.8$), we use the human judgment instead. Two authors independently make decisions, and have disagreements on approximately 5% of cases where a third author helps reach consensus. For high-confidence LLM judgments, we randomly sample 500 cases and only find 4 misjudgments (0.8% error rate), which validates the reliability of these LLM judgment. To ensure the consistency, we provide detailed instructions for the LLM and use the lowest temperature settings to minimize randomness. We also find that judgment inconsistency rarely happens across repeated invocations, and most of them occur in low-confidence cases.

Note that, as a generative AI, the LLM has non-deterministic outputs. Therefore, we store the original LLM/RAG output from the tested applications of each test to make sure that each schema repairs the same content.

6.1.5 Metrics. We evaluate in two dimensions.

Detection effectiveness: If a technique identifies a format/syntax/repetition error that leads to software misbehavior in a test, we refer to it as successfully *detecting the failure*, no matter the judgment of error type. We measure precision as the ratio of true positives (*i.e.*, correct detection) to all reported failures, and recall as the ratio of true positives to all actual failures.

Repairing effectiveness: If it converts the AI component output and eliminates the software misbehavior, we refer to it as successfully *preventing a potential failure*. We measure the execution pass rate after repair.

6.2 Answer to RQ1: Detection

As shown in Table 3, across 30,000 testing runs, Comfrey reports 11,754 errors, with 11,356 true positives (*i.e.*, the corresponding LLM/RAG output indeed has format/syntax/repetition errors) and only 398 false positives. The 11,356 true errors detected by Comfrey constitute 75.1% of all the 15,124 errors that occurred during the 30,000 testing runs.

Comfrey misses one quarter of integration failures due to two main reasons. Around a third of the false negatives are caused by dynamic software requirements, where the downstream components have requirements that change according to software internal state. For example, *AGiXT* [79] requires its task management agent to produce a template-based text response during user interaction, but a structured command when a tool invocation is required. As will be discussed in Section 7, this scenario is not targeted by Comfrey.

For the 398 false positives, the majority are wrong reports of format (37.2%), syntax (10.0%), and repetition (52.8%) errors. Particularly, 95.2% of false repetition reports stem from the incorrect redundant semantics detection, primarily due to the fundamental challenge of identifying the actual redundancy. First, embedding-based similarity analysis underperforms on domain-specific content, where semantically similar phrases serve distinct functional purposes. Second, it is inherently hard to distinguish necessary repetition (*e.g.*, recurring technical terms and the required semantic recurrence) from truly redundant content.

In comparison, the other three baselines either have many false negatives or false positives, or both.

For detection recall, *Retry*, *Reflexion*, *SmartGear*, and *JSON Schema Prompting* only detect 38, 6,234, 3,245, and 4,446 errors respectively, much less than what Comfrey detects. *Retry* performs poorly, as it is only able to detect non-silent failures. *Reflexion* has many false negatives, as LLMs are typically less capable of format validation. It lacks information across LLM invocations, which is essential for detecting repetition errors. *SmartGear* is only capable of tackling the string-related path-constraints, while many integration failures do not have such property. *JSON Schema Prompting* achieves moderate recall (29.4%) but is limited to format-related constraints and cannot address syntax parsing issues or semantic repetition errors that occur during runtime integration. Take *paper-qa* [76] in Section 3.3.3 as an example. Comfrey successfully detects the incorrect context construction error by identifying the semantic incoherence across content. However, *Retry* misses it as no exception is raised; *Reflexion* is only applied after LLM invocation; and *SmartGear* fails due to unawareness of coherence problem.

For detection precision, *Reflexion*, *SmartGear*, and *JSON Schema Prompting* have 8.0×, 3.2×, and 4.9× more false positives than Comfrey, respectively. Fully relying on LLM self-correction, *Reflexion* suffers severe hallucination due to its complete reliance on LLM self-correction, lacking external information for validation. *SmartGear* also has many mis-reports, because it assumes that AI components often make tiny errors and overly clusters their outputs to path constraints. *JSON Schema Prompting* frequently produces false positives when LLM outputs are semantically correct but deviate from rigid schema constraints, as it cannot distinguish between functionally equivalent formats and actual integration errors.

Summary: Comfrey achieves 75.1% recall and 96.6% precision in failure detection, significantly outperforming all baselines.

6.3 Answer to RQ2: Repair

As shown in Table 3 and Figure 6, Comfrey improves the execution correctness rate from 49.6% to 81.5%. It reports 11,754 errors and develops repair strategies for all of them. There are 62 repair attempts

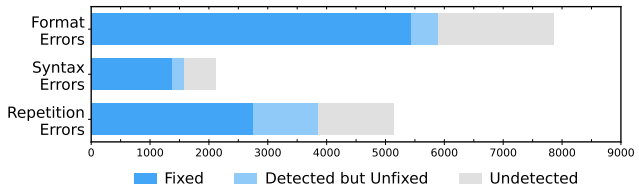

Figure 6: The number of original testing failures that can be fixed by different Comfrey strategies.

Table 4: Ablation study of Comfrey over each heuristic

	#(%) Detected Errors	#(%) Prevented Failures
Comfrey	11,356 (75.1%)	9,574 (63.3%)
(A) Remove resolving format errors	6,234 (41.2%)	3,768 (24.9%)
(B) Remove resolving syntax errors	10,832 (71.6%)	7,419 (49.0%)
(C) Remove resolving repetition errors	8,961 (59.3%)	5,747 (38.0%)

* Total # of original test failures is 15,124 (100%).

(0.2% of tests) that turn out to introduce errors to the correct AI outputs, of which the majority are caused by incorrect detection of repetition errors. Meanwhile, the remaining 11,356 repair attempts target truly problematic AI outputs and successfully repair 9,574 of them. As a net result, Comfrey reduces the number of integration failures from 15,124 to 5,550, resolving 68.8%/65.1%/53.2% of format/syntax/repetition errors.

In comparison, *Retry* only prevents 38 failures. Due to inaccurate failure detection and unreliable LLM self-correction mechanism, *Reflexion* only prevents one-third of the failures, improving execution correctness by 18.9%. *SmartGear* successfully repairs 62.4% of identified true errors, and improves execution correctness by 13.0%. *JSON Schema Prompting* prevents 4,446 failures (29.4%), improving execution correctness from 49.6% to 64.4% (14.8 percentage points improvement), but its preventive approach cannot address errors that emerge after LLM response generation. Having many detection false positives, *Reflexion*, *SmartGear*, and *JSON Schema Prompting* have 8×, 3×, and 4× more mis-repairs than Comfrey, respectively.

Take the intra-round repetition error in Figure 1 as an example. Comfrey successfully repairs the repetition errors by removing the redundant items (similarity=0.87). Meanwhile, *Retry* and *SmartGear* fail to identify this error. While *Reflexion* notices this error, its repair attempt unfortunately breaks the item order by rewriting the entire list. *JSON Schema Prompting* cannot prevent this error as it occurs during post-processing, showing the limitation of preventive approaches for integration failures.

Summary: Comfrey resolves 63.3% of the integration failures, improving software execution correctness from 49.6% to 81.5%.

6.4 Answer to RQ3: Ablation

We conduct an ablation study to understand the contribution of each error-tackling module in Comfrey. Table 4 shows the results when we remove one of the error tackling modules.

The format and repetition error tackling modules contribute the most to Comfrey. When removing the format(repetition) module, Comfrey prevents 60.6% (40.0%) fewer potential failures. This reflects the fact that format and repetition errors are the most common reasons for integration failures in LLM-enabled software, as discussed in Section 3.

The syntax error tackling module also has non-negligible contributions to Comfrey, preventing 22.5% of errors. The reason is the rareness of syntax errors, which only lead to 12% test failures. As shown in Figure 6, all modules share similar ratio of failure detection and prevention.

Summary: All 3 error-tackling modules are effective in preventing integration failures. The format/syntax/repetition module takes effect in preventing 60.6%/22.5%/40.0% of the potential failures.

6.5 Discussion

6.5.1 Sensitivity across apps. Comfrey shows similar detection and prevention capability across two application sets: the 50 applications used in our empirical study and the 50 additional applications. For the original set, Comfrey detects 5,724 (75.4%) and prevents 4,803 (63.3%) out of 7,593 testing failures. For the additional set, Comfrey detects 5,632 (74.8%) and prevents 4,771 (63.4%) out of 7,531 testing failures. Comfrey improves the execution correctness rate from 49.6% to 81.5% for both sets. This result demonstrates the generalizability of Comfrey.

6.5.2 Performance overhead. Comfrey always attempts to use lightweighted techniques (*e.g.*, AST parsing) to tackle integration failures. The main overhead comes from invoking embedding models through cloud APIs, which is invoked in 65.2% tests. Such invocation typically takes 1.2-2.8 seconds, and Comfrey uses batch processing to reduce overhead. Across all 30,000 test cases, Comfrey introduces 8.4% performance overhead on average. When there is no such invocation, the overhead is around 2.1%.

In comparison, *Retry* has neglectable overhead, due to invocation in less than 0.3% tests. *SmartGear* has 1.2% overhead, as it only involves rule-based string operations. *Reflexion* requires multiple LLM API calls for iterative correction and incurs 156.3% overhead on average across all test cases.

6.5.3 Sensitivity analysis of τ threshold. We conduct a sensitivity analysis for the semantic similarity threshold $\tau = 0.7$ used in both incorrect context construction detection (§4.3.3) and redundant semantics detection (§4.5.2).

We evaluate Comfrey's performance across different τ values ranging from 0.5 to 0.9 with 0.1 increments. Performance remains relatively stable across applications, with average precision varying between 94.2%-97.8% and recall between 73.9%-76.8%. Due to this insensitivity, we adopt the median value $\tau = 0.7$ as the default setting of Comfrey.

6.5.4 User study. To further evaluate the repairs of Comfrey, we have conducted a user study with 70 participants who volunteered to answer a software-quality survey. The survey contains 12 repair attempts randomly selected from 12 applications, covering format, syntax, and repetition errors. For each repair attempts, the survey

provides a brief introduction of the application, the original software output and the repaired one, and three 5-point Likert scale questions about: (1) whether they are satisfied with the original software output, (2) whether they are satisfied with the software output after repair, and (3) whether the repaired software output preserves the semantics of original one. The participants were invited through email invitations from 8 academic institutions. Participants are aged 18–55, with around 60% being male. Most of them have computer science background, and around 80% have experience with LLM-enabled applications or AI integration.

As shown in Table 5, most participants are more satisfied with the repaired software output, with 2.93 points of satisfactory improvement on average. Particularly, 85% participants prefer Comfrey's repair attempts in *all cases*. In terms of semantic preservation, participants dominantly agree that Comfrey's repair attempts do not change the semantics of original software output.

Table 5: User study results on repair quality evaluation

Evaluation Aspect	Error Type			Average
	Format	Syntax	Repetition	
Original Output Satisfaction	1.85	1.88	1.93	1.89
Repaired Output Satisfaction	4.79	4.84	4.83	4.82
Improvement (Δ)	2.94	2.97	2.90	2.93
Semantic Preservation	4.82	4.84	4.85	4.84

* Scores refer to satisfactory based on 5-point Likert scale evaluation.

7 Thread to Validity

Internal threats to validity. Comfrey assumes the specifications of AI component output are completely characterized by the downstream tasks and application scenarios, which could be incorrect. The stage-wise error-tackling design of Comfrey assumes that the failure caused by combined errors has all the corresponding symptoms, which is not guaranteed. The static analysis module of Comfrey cannot capture the requirements that emerge dynamically during runtime and domain-specific semantic constraints.

External threats to validity. Comfrey is only evaluated with Python applications that incorporate LLM agents of four representative types, which may not represent all real-world LLM-enabled software. The test inputs are created from public AI datasets, which may be biased and not cover all possible run-time scenarios.

8 Related Work

8.1 LLM-enabled Software

Several work conducts empirical studies of LLM-enabled software from the perspectives of users [108], developers [109], and platforms [110], emphasizing their rapid growth. Some work [111, 112] studies user privacy, harmful content, and security vulnerabilities of LLM-enabled software. Another line of work conducts empirical study of defects [63, 113] in real-world LLM-enabled applications.

These work aims to understand the current situation and challenges of LLM-enabled software. They only provide general guidelines obtained through manual code inspection. In contrast, Comfrey proposes an *automated* approach for preventing the failures.

8.2 AI-related Testing and Fixing

Recent work identifies the intrinsic defects in LLMs, including hallucinations [114], context understanding challenges [31–33], and safety vulnerabilities [115–117]. Several benchmarks [118–121] are proposed for evaluating LLMs in various domains and perspectives Another line of work proposes automated solutions for testing [43–45, 122] and fixing [123–127] LLMs. They focus on LLM itself and do not tackle the system integration challenges.

Some work improves LLM capability on specific tasks through prompt engineering and output post-processing, including software engineering [50], scientific discovery [52], and clinical decision support [53, 55, 128]. These works are task-specific and cannot tackle the various requirements in LLM-enabled software.

Much work analyzes AI-driven software at the system level, tackling integration challenges of traditional machine learning API [56, 129–131], and of deep learning models [132–135]. They are designed for AI models with pre-defined tasks and categorical outputs, instead of LLMs. Other research on testing [132, 136–138] and fixing [139–143] neural networks are orthogonal to our work.

8.3 LLM Output Constraint Approaches

Recently, several approaches aim to constrain and validate LLM outputs for specific requirements. Constrained decoding [144] inspects intermediate inference results to enforce lexical constraints during text generation. However, it could only tackle format errors and only applicable to open-source LLMs that are deployed locally. Additionally, existing work like Guardrails [145] and Outlines [146] focuses on runtime output validation and constraint enforcement. They propose novel architectures and programming paradigms, requiring huge human effort on software refactoring. They are also limited to format errors. Conversely, Comfrey addresses a wider range of integration failures with plug-and-play solutions.

9 Conclusion

LLM-enabled software encounters integration challenges when the output of AI component violates the format, syntax, and repetition requirements of its downstream software task. In this paper, we present Comfrey, a runtime framework that prevents integration failures in LLM-enabled software. Serving as a middle layer between AI and non-AI components, Comfrey automatically detects and repairs incompatible AI component outputs via a three-stage workflow. We evaluate a variety of open-source applications to demonstrate their effectiveness and efficiency.

Acknowledgement

This paper is supported by National Natural Science Foundation of China (Grant No. 62402183, 92582108, 62572192), the Chenguang Program of Shanghai Education Development Foundation and Shanghai Municipal Education Commission (Grant 23CGA33), the Shanghai Special Program for Promoting High-Quality Industrial Development (Project No. 250203, 250668), and Shanghai Trusted Industry Internet Software Collaborative Innovation Center. This work is also supported in part by JST CRONOS Grant (No. JPMJCS24K8), JSPS KAKENHI Grant (No.JP21H04877, No.JP23H03372, and No.JP24K02920), Canada CIFAR AI Chairs Program, the Natural Sciences and Engineering Research Council of Canada.

References

[1] Y. Shao, "Comfrey." https://github.com/ycshao12/Comfrey_2026, 2026.
[2] lencx, "Chatgpt desktop application." https://github.com/lencx/ChatGPT, 2022.
[3] C. AI, "Chatbox," 2023.
[4] Reworkd, "Agentgpt: Autonomous ai agents in your browser," 2023.
[5] josStorer, "chatgptbox," 2022.
[6] Y. Nakajima, "Babyagi," 2023.
[7] T. Kipkemboi, "Trip planner agent: Ai-powered travel itinerary generator," 2023.
[8] DoggoOP, "Personalizedstudyplanner: Ai-powered study planning tool," 2023.
[9] hkust zhiyao, "Rtl-coder," 2024.
[10] Damarcreative, "Codesheek: Ai-powered natural-language-to-web-project code generator," 2023.
[11] V. Rudloff, "Appifyai: Transform conversations into stunning web apps," 2023.
[12] D. Huang, T.-S. Nguyen, and *et al.*, "RAP: A metric for balancing repetition and performance in open-source large language models," in *NAACL*, pp. 1479–1496, 2025.
[13] Z. Wang, J. Jiang, and *et al.*, "Verifiable format control for large language model generations," in *NAACL*, pp. 3499–3513, 2025.
[14] H. Wen, Y. Zhu, C. Liu, X. Ren, W. Du, and M. Yan, "Fixing function-level code generation errors for foundation large language models," 2025.
[15] F. Hassan, N. Meng, and X. Wang, "Uniloc: Unified fault localization of continuous integration failures," *TOSEM*, vol. 32, no. 6, 2023.
[16] F. Hassan, "Tackling build failures in continuous integration," *ASE*, pp. 1242–1245, 2019.
[17] M. Cataldo and J. D. Herbsleb, "Factors leading to integration failures in global feature-oriented development: an empirical analysis," *ICSE*, pp. 161–170, 2011.
[18] M. Santolucito, J. Zhang, E. Zhai, J. Cito, and R. Piskac, "Learning CI configuration correctness for early build feedback," in *SANER'22*, pp. 1006–1017, IEEE.
[19] C. Wan, S. Liu, H. Hoffmann, M. Maire, and S. Lu, "A replication of are machine learning cloud apis used correctly," in *ICSE*, 2021.
[20] L. Chen, M. Zaharia, and J. Zou, "Frugalml: how to use ml prediction apis more accurately and cheaply," in *NeurIPS*, 2020.
[21] L. Chen, M. A. Zaharia, and J. Y. Zou, "Efficient online ml api selection for multi-label classification tasks," in *ICML*, 2021.
[22] Q. Huang, X. wen Dong, P. Zhang, *et al.*, "Opera: Alleviating hallucination in multi-modal large language models via over-trust penalty and retrospection-allocation," *CVPR*, pp. 13418–13427, 2023.
[23] T. Yu, Y. Yao, H. Zhang, T. He, *et al.*, "Rlhf-v: Towards trustworthy mllms via behavior alignment from fine-grained correctional human feedback," *CVPR*, pp. 13807–13816, 2023.
[24] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica, "Judging llm-as-a-judge with mt-bench and chatbot arena," in *NeurIPS*, 2023.
[25] G. Cui, L. Yuan, N. Ding, G. Yao, B. He, W. Zhu, Y. Ni, G. Xie, R. Xie, Y. Lin, Z. Liu, and M. Sun, "Ultrafeedback: boosting language models with scaled ai feedback," in *ICML'24*, 2024.
[26] D. Zhang, S. Zhoubian, Z. Hu, Y. Yue, Y. Dong, and J. Tang, "Rest-mcts*: Llm self-training via process reward guided tree search," *NeurIPS*, vol. 37, pp. 64735–64772, 2024.
[27] A. Setlur, S. Garg, X. Geng, N. Garg, V. Smith, and A. Kumar, "Rl on incorrect synthetic data scales the efficiency of llm math reasoning by eight-fold," *NeurIPS*, vol. 37, pp. 43000–43031, 2024.
[28] S. Deng, W. Xu, H. Sun, W. Liu, T. Tan, L. Liujianfeng, A. Li, J. Luan, B. Wang, R. Yan, and S. Shang, "Mobile-bench: An evaluation benchmark for LLM-based mobile agents," in *ACL*, pp. 8813–8831, 2024.
[29] M. Katz, H. Kokel, K. Srinivas, and S. Sohrabi, "Thought of search: Planning with language models through the lens of efficiency," in *NeurIPS*, 2024.
[30] Y. Li, Y. Huang, and *et al.*, "Snapkv: Llm knows what you are looking for before generation," *NeurIPS*, vol. 37, pp. 22947–22970, 2024.
[31] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," *TACL*, vol. 12, pp. 157–173, 2024.
[32] Z. Zhang, R. Chen, S. Liu, Z. Yao, O. Ruwase, B. Chen, X. Wu, and Z. Wang, "Found in the middle: How language models use long contexts better via plug-and-play positional encoding," in *NeurIPS*, vol. 37, 2024.
[33] S. An, Z. Ma, Z. Lin, N. Zheng, and J.-G. Lou, "Make your LLM fully utilize the context," in *NeurIPS*, vol. 37, 2024.
[34] J. Jin, Y. Zhu, Y. Zhou, and Z. Dou, "BIDER: Bridging knowledge inconsistency for efficient retrieval-augmented LLMs via key supporting evidence," in *ACL* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), pp. 750–761, ACL, 2024.
[35] Y. Liu, X. Peng, X. Zhang, W. Liu, J. Yin, J. Cao, and T. Du, "RA-ISF: Learning to answer and understand from retrieval augmentation via iterative self-feedback," in *ACL* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), pp. 4730–4749, ACL, 2024.
[36] R. McIlroy-Young, K. Brown, C. Olson, L. Zhang, and C. Dwork, "Order-independence without fine tuning," *NeurIPS*, vol. 37, pp. 72818–72839, 2024.
[37] J. Cao, M. Li, M. Wen, and S.-C. Cheung, "A study on prompt design, advantages and limitations of chatgpt for deep learning program repair," *ASE*, vol. 32, no. 1, 2025.
[38] M. U. Khattak, H. A. Rasheed, M. Maaz, S. H. Khan, and F. S. Khan, "Maple: Multi-modal prompt learning," *CVPR*, pp. 19113–19122, 2022.
[39] S. Roy and A. Etemad, "Consistency-guided prompt learning for vision-language models," in *ICLR*, 2024.
[40] H. Liu, C. Li, Q. Wu, and Y. J. Lee, "Visual instruction tuning," in *NIPS*, 2023.
[41] J. Li, D. Li, and *et al.*, "Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models," in *ICML*, 2023.
[42] W. Dai, J. Li, and *et al.*, "Instructblip: towards general-purpose vision-language models with instruction tuning," in *NIPS*, 2023.
[43] N. Mündler, M. N. Müller, J. He, and M. T. Vechev, "Swt-bench: Testing and validating real-world bug-fixes with code agents," in *NeurIPS*, 2024.
[44] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *ICSE*, ACM, 2024.
[45] S. Zhang, H. Zhao, X. Liu, Q. Zheng, Z. Qi, X. Gu, Y. Dong, and J. Tang, "Naturalcodebench: Examining coding performance mismatch on humaneval and natural user queries," in *ACL*, 2024.
[46] J. Zhang, T. Mytkowicz, M. Kaufman, R. Piskac, and S. K. Lahiri, "Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)," in *ISSTA* (S. Ryu and Y. Smaragdakis, eds.), 2022.
[47] A. Maharana, D.-H. Lee, S. Tulyakov, M. Bansal, F. Barbieri, and Y. Fang, "Evaluating very long-term conversational memory of LLM agents," in *ACL*, pp. 13851–13870, 2024.
[48] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and W. B. Dolan, "Dialogpt : Large-scale generative pre-training for conversational response generation," in *ACL*, 2019.
[49] E. M. Smith, M. Williamson, and *et al.*, "Can you put it all together: Evaluating conversational agentsâĂŹ ability to blend skills," in *ACL*, 2020.
[50] E. Chamoun, M. Schlichtkrull, and A. Vlachos, "Automated focused feedback generation for scientific writing assistance," in *ACL*, 2024.
[51] Z. Yang, Z. Zhou, and *et al.*, "MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization," in *ACL*, pp. 11789–11804, 2024.
[52] X. Hu, Z. Zhao, S. Wei, Z. Chai, Q. Ma, *et al.*, "Infiagent-dabench: evaluating agents on data analysis tasks," in *ICML*, 2024.
[53] Y. Labrak, A. Bazoge, E. Morin, P.-A. Gourraud, M. Rouvier, and R. Dufour, "Biomistral: A collection of open-source pretrained large language models for medical domains," in *ACL*, 2024.
[54] X. Huang, L. Shen, J. Liu, F. Shang, and *et al.*, "Towards a multimodal large language model with pixel-level insight for biomedicine," in *AAAI*, 2025.
[55] J. Vladika, P. Schneider, and F. Matthes, "MedREQAL: Examining medical knowledge recall of large language models via question answering," in *ACL* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), pp. 14459–14469, ACL, 2024.
[56] 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," *OOPSLA*, vol. 7, 2023.
[57] J. Liu, K. Wang, Y. Chen, X. Peng, and *et al.*, "Large language model-based agents for software engineering: A survey," *arXiv:2409.02977*, 2024.
[58] Z. Rasool, S. Barnett, S. Kurniawan, S. Balugo, R. Vasa, C. Chesser, and A. Bahar-Fuchs, "Evaluating llms on document-based qa: Exact answer selection and numerical extraction using cogtale dataset," *ArXiv*, vol. abs/2311.07878, 2023.
[59] L. AI, "Chat langchain." https://github.com/langchain-ai/chat-langchain.
[60] D. Patel, "Lightgpt: A lightweight implementation of chatgpt using transformers." https://github.com/darshit001/LightGPT, 2024.
[61] QuivrHQ, "Quivr: Your second brain, powered by ai." https://github.com/QuivrHQ/quivr, 2023. Accessed: 2024-03-01.
[62] H2O.ai, "h2ogpt: Open-source generative ai platform," 2023.
[63] Y. Shao, Y. Huang, J. Shen, L. Ma, T. Su, and C. Wan, "Are llms correctly integrated into software systems?," in *ICSE*, 2025.
[64] Q. Team, "Qwen: Large language models by alibaba cloud." https://github.com/QwenLM/Qwen, 2023. Accessed: 2025-06-23.
[65] P. Budzianowski, T.-H. Wen, B.-H. Tseng, I. Casanueva, S. Ultes, O. Ramadan, and M. Gašić, "MultiWOZ - a large-scale multi-domain Wizard-of-Oz dataset for task-oriented dialogue modelling," in *EMNLP*, pp. 5016–5026, ACL, 2018.
[66] B. Byrne, K. Krishnamoorthi, C. Sankar, A. Neelakantan, B. Goodrich, D. Duckworth, S. Yavuz, A. Dubey, K.-Y. Kim, and A. Cedilnik, "Taskmaster-1: Toward a realistic and diverse dialog dataset," in *EMNLP-IJCNLP* (K. Inui, J. Jiang, V. Ng, and X. Wan, eds.), pp. 4516–4525, ACL, 2019.
[67] H. Hüttel, "On program synthesis and large language models," *Commun. ACM*, vol. 68, p. 33âĂŞ35, Dec. 2024.
[68] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, "Ds-1000: a natural and reliable benchmark for data science code generation," in *ICML*, JMLR.org, 2023.
[69] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, and *et al.*, "Measuring coding challenge competence with apps (2021)," *arXiv:2105.09938*, 2021.
[70] T. Kočiský, J. Schwarz, P. Blunsom, C. Dyer, K. M. Hermann, G. Melis, and E. Grefenstette, "The NarrativeQA reading comprehension challenge," *TACL*,

vol. 6, pp. 317–328, 2018.
[71] BBC, "Bbc - homepage," 2025. Accessed: 2025-06-20.
[72] arXiv, "arxiv e-print archive," 2025. Accessed: 2025-06-20.
[73] J. Gu, X. Jiang, Z. Shi, H. Tan, X. Zhai, C. Xu, W. Li, Y. Shen, S. Ma, and *et al.*, "A survey on llm-as-a-judge," 2025.
[74] Abonia, "Context-based-llmchatbot." https://github.com/Abonia1/Context-Based-LLMChatbot, 2023.
[75] S. Zhao, Y. Huang, J. Song, Z. Wang, C. Wan, and L. Ma, "Towards understanding retrieval accuracy and prompt quality in rag systems," 2024.
[76] F. House, "Paper-qa: Ask questions about your papers in natural language." https://github.com/Future-House/paper-qa, 2023. Accessed: 2025-07-10.
[77] J. Shen, C. Wan, R. Qiao, J. Zou, H. Xu, Y. Shao, Y. Zhang, W. Miao, and G. Pu, "A study of in-context-learning-based text-to-sql errors," 2025.
[78] Eosphoros-ai, "Db-gpt: Revolutionizing database interactions with private llm technology." https://github.com/eosphoros-ai/DB-GPT, 2023.
[79] J. XT, "Agixt: An artificial general intelligence automation platform." https://github.com/Josh-XT/AGiXT, 2023.
[80] FSA, "Finite state automata - CS field guide," 2024.
[81] A. Páez and G. Boisjoly, *Exploratory Data Analysis*, pp. 25–64. Cham: Springer International Publishing, 2022.
[82] A. Fang, C. Macdonald, I. Ounis, and P. Habel, "Using word embedding to evaluate the coherence of topics from twitter data," in *SIGIR*, (New York, NY, USA), p. 1057âĂŞ1060, Association for Computing Machinery, 2016.
[83] G. Bedi, F. Carrillo, G. A. Cecchi, and *et al.*, "Automated analysis of free speech predicts psychosis onset in high-risk youths," *npj Schizophrenia*, vol. 1, no. 1, pp. 1–7, 2015.
[84] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.
[85] Y. Yu, W. Ping, Z. Liu, B. Wang, J. You, C. Zhang, M. Shoeybi, and B. Catanzaro, "RankRAG: Unifying context ranking with retrieval-augmented generation in LLMs," in *NeurIPS*, 2024.
[86] Unicode Consortium, "The unicode standard, version 15.1.0," 2024.
[87] W. B. Cavnar and J. M. Trenkle, "N-gram-based text categorization," in *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, pp. 161–175, 1994.
[88] H. T. Ng, S. M. Wu, T. Briscoe, C. Hadiwinoto, R. H. Susanto, and C. Bryant, "The conll-2014 shared task on grammatical error correction," in *18th CoNLL: Shared Task*, pp. 1–14, 2014.
[89] O. Topsakal and T. C. Akinci, "Creating large language model applications utilizing langchain: A primer on developing llm apps fast," in *ICAENS*, vol. 1, pp. 1050–1056, 2023.
[90] H. Chase, "Langchain: Building applications with llms through composability." https://github.com/langchain-ai/langchain, 2022.
[91] ESLint, "Eslint," 2025.
[92] SonarSource, "Sonarqube," 2025.
[93] Python, "Python 3.13.0 documentation: Glossary - decorator," 2025.
[94] V. Stinner, "Bytecode: Python module to generate and modify bytecode." https://pypi.org/project/bytecode/, 2021. Online document.
[95] Python, "The python tutorial: 1.1. invoking the interpreter." https://docs.python.org/3/tutorial/interpreter.html, 2024. Accessed: 2025-07-04.
[96] Technologicat, "pyan: Static analyzer for python that generates call graphs." https://github.com/Technologicat/pyan, 2024. Accessed: 2025-07-04.
[97] D. Halter, "Jedi: An autocompletion tool for python." https://github.com/davidhalter/jedi, 2024. Accessed: 2025-07-04.
[98] Python, *ast — Abstract Syntax Trees*, 2024. Accessed: 2025-07-04.
[99] A. A. Hagberg, D. A. Schult, P. J. Swart, and N. Developers, "Networkx: Python software for complex networks." https://github.com/networkx/networkx, 2024.
[100] S. S. Paille, "Beniget: Static analysis of python bindings using dataflow graphs." https://github.com/serge-sans-paille/beniget, 2024.
[101] Python, *re — Regular expression operations*, 2023. Python 3.12.0 documentation.
[102] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. O'Reilly Media, Inc., 2009.
[103] F. Pedregosa, G. Varoquaux, and G. *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Nov. 2011.
[104] N. Developers, "Numpy: The fundamental package for scientific computing with python." https://github.com/numpy/numpy, 2024. Accessed: 2025-07-02.
[105] N. Shinn, F. Cassano, A. Gopinath, K. R. Narasimhan, and S. Yao, "Reflexion: language agents with verbal reinforcement learning," in *NeurIPS*, 2023.
[106] B. T. Willard and R. Louf, "Efficient guided generation for large language models," 2023.
[107] A. Kumar, R. Morabito, S. Umbet, J. Kabbara, and A. Emami, "Confidence under the hood: An investigation into the confidence-probability alignment in large language models," in *ACL*, pp. 315–334, 2024.
[108] Y. Zhao, X. Hou, S. Wang, and H. Wang, "Llm app store analysis: A vision and roadmap," *TOSEM*, vol. 34, no. 5, pp. 1–25, 2025.
[109] X. Chen, C. Gao, C. Chen, G. Zhang, and Y. Liu, "An empirical study on challenges for llm application developers," *TOSEM*, 2025.
[110] K. Hau, S. Hassan, and S. Zhou, "Llms in mobile apps: Practices, challenges, and opportunities," in *MOBILESoft*, pp. 3–14, IEEE, 2025.
[111] T. Liu, Z. Deng, G. Meng, Y. Li, and K. Chen, "Demystifying rce vulnerabilities in llm-integrated apps," in *CCS*, pp. 1716–1730, 2024.
[112] X. Hou, Y. Zhao, and H. Wang, "On the (in) security of llm app stores," in *2025 IEEE Symposium on Security and Privacy (SP)*, pp. 317–335, IEEE, 2025.
[113] W. Fan, Y. Ding, L. Ning, and *et al.*, "A survey on rag meeting llms: Towards retrieval-augmented large language models," in *KDD*, ACM, 2024.
[114] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, and *et al.*, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, 2025.
[115] X. Qi, Y. Zeng, T. Xie, and *et al.*, "Fine-tuning aligned language models compromises safety, even when users do not intend to!," in *ICLR*, 2024.
[116] R. Staab, M. Vero, M. Balunovic, and M. Vechev, "Beyond memorization: Violating privacy via inference with large language models," in *ICLR*, 2024.
[117] Z. Xiang, F. Jiang, Z. Xiong, B. Ramasubramanian, R. Poovendran, and B. Li, "Badchain: Backdoor chain-of-thought prompting for large language models," in *NeurIPS*, 2024.
[118] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *NeurIPS*, 2023.
[119] Y. Ding, Z. Wang, W. U. Ahmad, *et al.*, "Crosscodeeval: a diverse and multilingual benchmark for cross-file code completion," in *NeurIPS*, 2023.
[120] M. Joshi, E. Choi, D. Weld, and L. Zettlemoyer, "Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension," in *ACL*, 2017.
[121] B. Wang, W. Chen, and *et al.*, "Decodingtrust: A comprehensive assessment of trustworthiness in gpt models.," in *NeurIPS*, 2023.
[122] Y. Huang, J. Song, and *et al.*, "Active testing of large language model via multi-stage sampling," *arXiv:2408.03573*, 2024.
[123] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," in *ICLR*, 2024.
[124] Y. Huang, L. Ma, K. Nishikino, and T. Akazaki, "Risk assessment framework for code llms via leveraging internal states," in *ESEC/FSE*, pp. 1314–1325, 2025.
[125] Y. Shi, S. Wang, C. Wan, and X. Gu, "From code to correctness: Closing the last mile of code generation with hierarchical debugging," 2024.
[126] L. Pan, M. Saxon, W. Xu, D. Nathani, X. Wang, and W. Y. Wang, "Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies," *TACL*, vol. 12, pp. 484–506, 2024.
[127] A. Madaan *et al.*, "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
[128] J. He, Y. Gong, Z. Lin, C. Wei, Y. Zhao, and K. Chen, "Llm factoscope: Uncovering llms' factual discernment through measuring inner states," in *ACL*, 2024.
[129] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Keeper: Automated testing and fixing of machine learning software," *TOSEM*, 2024.
[130] C. Wan, S. Liu, S. Xie, Y. Liu, H. Hoffmann, M. Maire, and S. Lu, "Automated testing of software that uses machine learning apis," in *ICSE*, 2022.
[131] R. Wu, C. Guo, A. Hannun, and L. van der Maaten, "Fixes that fail: self-defeating improvements in machine-learning systems," in *NeurIPS*, 2021.
[132] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, Y. Wang, and X. Zhang, "A survey of testing deep learning systems," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–38, 2020.
[133] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *ICSE*, 2023.
[134] P. Pan, S. Swaroop, A. Immer, R. Eschenhagen, R. E. Turner, and M. E. Khan, "Continual deep learning by functional regularisation of memorable past," in *NeurIPS*, Curran Associates Inc., 2020.
[135] G. Jahangirova, N. Humbatova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *ICSE*, 2020.
[136] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: automated testing of deep-neural-network-driven autonomous cars," in *ICSE*, 2018.
[137] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *ICML*, 2019.
[138] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S.-C. Cheung, and T. Xie, "Detecting numerical bugs in neural network architectures," in *ESEC/FSE*, 2020.
[139] R. Tanno, M. F. Pradier, A. Nori, and Y. Li, "Repairing neural networks by leaving the right past behind," in *NeurIPS*, Curran Associates Inc., 2022.
[140] H. Zhang and W. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *ASE*, 2019.
[141] Z. Li, X. Ma, C. Xu, J. Xu, C. Cao, and *et al.*, "Operational calibration: Debugging confidence errors for dnns in the field," in *ESEC/FSE*, 2020.
[142] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *ICSE*, 2020.
[143] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *ICSE*, 2020.
[144] C. Hokamp and Q. Liu, "Lexically constrained decoding for sequence generation using grid beam search," 2017.
[145] G. AI, "Guardrails," 2025. Accessed: 2025-10-25.
[146] D. AI, "Outlines," 2025. Accessed: 2025-10-25.