LADYBUG: an LLM Agent DeBUGger for data-driven applications

Joel Rorseth
University of Waterloo
Waterloo, Canada
jerorset@uwaterloo.ca

Parke Godfrey
York University
Toronto, Canada
godfrey@yorku.ca

Lukasz Golab
University of Waterloo
Waterloo, Canada
lgolab@uwaterloo.ca

Divesh Srivastava
AT&T Chief Data Office
New Jersey, USA
divesh@research.att.com

Jarek Szlichta
York University
Toronto, Canada
szlichta@yorku.ca

ABSTRACT

We demonstrate LADYBUG, an interactive tool designed for tracing and debugging the outputs of large language model (LLM) agents in data-driven applications. LADYBUG enables users to trace the steps executed by an agent, intervene at arbitrary steps, and efficiently re-execute affected steps. To help debug important but inconspicuous issues, we implement an LLM-aided debugger that leverages self-reflection to identify incorrect steps and propose efficient interventions.

1 INTRODUCTION

Motivation. Advances in large language models (LLMs) have transformed natural language into a powerful mode for programming. LLMs are prone to critical lapses, however, with hallucinations and mathematical inaccuracies. These can be difficult to debug [4]. To mitigate these issues, data scientists are now building *LLM agents* which solve complex multi-step problems by coordinating calls to LLMs, tools, and other operators. While these agents alleviate some of the weaknesses of individual LLMs, debugging them is further complicated by the introduction of programmatic constructs, such as control flow and tool invocations, whose errors can compound across steps. That is, LLM agents have evolved to mirror traditional software programs, with a clear need for specialized debugging tools akin to traditional software debuggers.

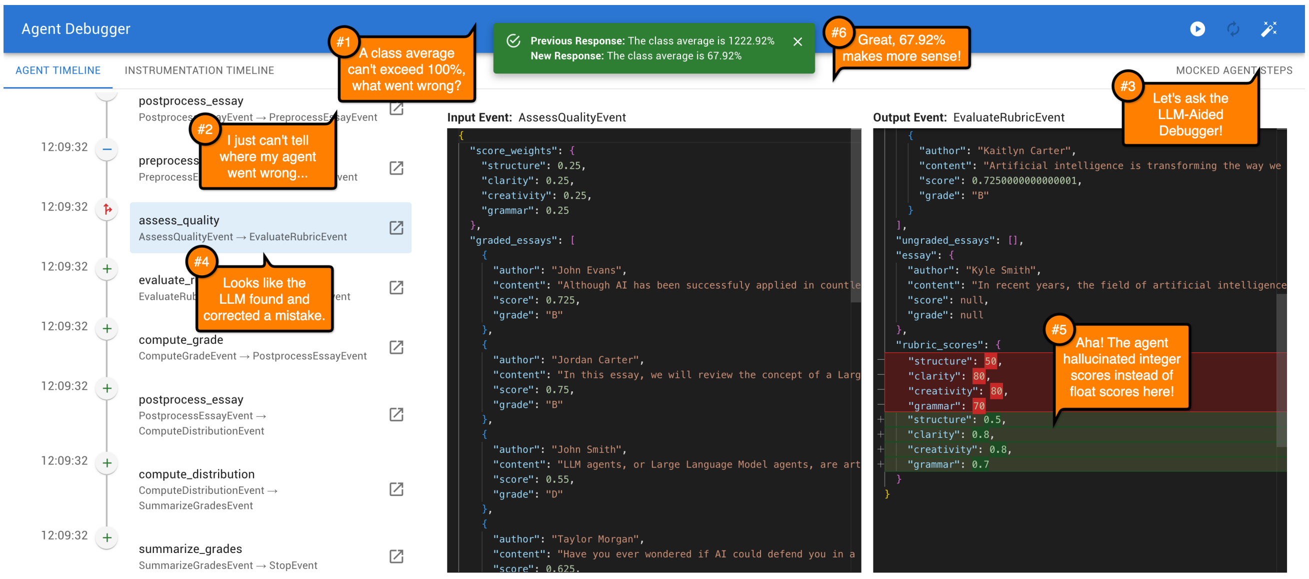
Debugging these LLM agents presents challenges similar to those encountered in conventional software development. Like software programs, agents execute sequences of operations with arbitrary inputs and outputs [1]. Each step may affect those that follow. Software debuggers often enable stepwise reenactment of an execution *trace*, allowing users to backtrack through previous steps, identify errors, modify state, and test potential fixes. LLM agents lack any analogous tool that enables inspection and intervention at intermediate steps. Thus, users are only left with the inefficient option of repeatedly modifying and re-executing the entire agent.

LLM agents can be debugged using existing software debuggers, but fundamental differences in their design necessitate dedicated tooling. Software debuggers operate at a low-level: stepping through individual *lines of code* and modifying in-memory variables. The atomic units of computation in an agent—i.e., their steps and their inputs / outputs—are higher-level constructs, however. In this sense, debugging an agent is more akin to debugging a data pipeline [2], as the atomic units of computation are often black-box *modules* rather than atomic instructions. Thus, observing execution traces, backtracking, or intervening on an LLM agent is cumbersome and inefficient using a regular software debugger. Furthermore, agent steps are fundamentally more complex than individual lines in a software program, as they define sequences of potentially expensive operations (e.g., LLM calls and data retrieval) [6]. It is therefore important for agents to avoid unnecessary re-execution. However, traditional software debuggers do not facilitate "random access" intervention at arbitrary agent steps, nor do they spot issues or suggest efficient interventions to fix them.

Contributions. To enable these capabilities and more, we demonstrate LADYBUG,[1] a novel framework for tracing and debugging the outputs of LLM agents,[2] such as those built with LlamaIndex, LangChain, and CrewAI. We implement a novel debugging tool that enables users to trace, modify, and efficiently re-execute the intermediate steps of an LLM agent, facilitating diagnosis and repair of hallucinations and other issues. We note that, intriguingly, LLMs hold the potential to address issues in their own output or that of other LLMs. By leveraging *self-reflection*, LLMs can be guided to critique and refine their outputs, mitigating the very hallucinations they produce. We therefore implement a novel LLM-aided agent debugger, which helps identify incorrect agent steps and propose interventions that avoid unnecessary re-execution.

To illustrate, consider an LLM agent that helps teachers grade student essays, formalized in Section 3.3 as Use Case #3. This data-driven agent performs several steps, starting by ingesting the essays, assessing writing quality, computing grades, and summarizing the class grade distribution. As illustrated in Figure 1, the user can click the "play" button in the navigation bar of our tool to execute the agent, which, upon completion, renders a chronological trace of executed steps (left) under the "Agent Timeline" tab. Suppose that the success alert (callout #1) indicates that an anomalous class average was returned. After reviewing the input and output of each step (the code views to the right of the trace), however, the user cannot find any issues (#2). The user clicks the "assist" button in the navigation bar (#3) to enlist the help of the LLM-aided debugger, which prompts an LLM to identify and correct an LLM hallucination occurring in a late step. The interface updates to show the corrected step in the

[1]A video is available at https://vimeo.com/1036113298.
[2]The tool is available at http://lg-research-2.uwaterloo.ca:8091/ladybug.


Agent Debugger
AGENT TIMELINE
INSTRUMENTATION TIMELINE
MOCKED AGENT STEPS
#1 A class average can't exceed 100%, what went wrong?
#2 I just can't tell where my agent went wrong...
#3 Let's ask the LLM-Aided Debugger!
#4 Looks like the LLM found and corrected a mistake.
#5 Aha! The agent hallucinated integer scores instead of float scores here!
#6 Great, 67.92% makes more sense!

postprocess_essay
Postproces… …ayEvent → PreprocessE… …sayEvent

12:09:32 preprocess…
PreprocessE… …vent

12:09:32 assess_quality
AssessQualityEvent → EvaluateRubricEvent

12:09:32 evaluate_r…
EvaluateRub… …ent

12:09:32 compute_grade
ComputeGradeEvent → PostprocessEssayEvent

12:09:32 postprocess_essay
PostprocessEssayEvent → ComputeDistributionEvent

12:09:32 compute_distribution
ComputeDistributionEvent → SummarizeGradesEvent

12:09:32 summarize_grades
SummarizeGradesEvent → StopEvent

Previous Response: The class average is 1222.92%
New Response: The class average is 67.92%

Input Event: AssessQualityEvent

```
{
  "score_weights": {
    "structure": 0.25,
    "clarity": 0.25,
    "creativity": 0.25,
    "grammar": 0.25
  },
  "graded_essays": [
    {
      "author": "John Evans",
      "content": "Although AI has been successfuly applied in countle
      "score": 0.725,
      "grade": "B"
    },
    {
      "author": "Jordan Carter",
      "content": "In this essay, we will review the concept of a Larg
      "score": 0.75,
      "grade": "B"
    },
    {
      "author": "John Smith",
      "content": "LLM agents, or Large Language Model agents, are art
      "score": 0.55,
      "grade": "D"
    },
    {
      "author": "Taylor Morgan",
      "content": "Have you ever wondered if AI could defend you in a
      "score": 0.625,
```

Output Event: EvaluateRubricEvent

```
      {
        "author": "Kaitlyn Carter",
        "content": "Artificial intelligence is transforming the way we
        "score": 0.7250000000000001,
        "grade": "B"
      }
    ],
    "ungraded_essays": [],
    "essay": {
      "author": "Kyle Smith",
      "content": "In recent years, the field of artificial intelligence
      "score": null,
      "grade": null
    },
    "rubric_scores": {
-     "structure": 50,
-     "clarity": 80,
-     "creativity": 80,
-     "grammar": 70
+     "structure": 0.5,
+     "clarity": 0.8,
+     "creativity": 0.8,
+     "grammar": 0.7
    }
  }
```

Figure 1: Using LADYBUG to debug an agent that helps teachers grade essays but which returned an anomalous class average.

timeline (denoted by the red "fork" icon in #4) and its substituted output (denoted by green highlighter in #5), then re-executes the subsequent steps to obtain a new class average (#6).

Our main contributions are as follows.

- **LLM Agent Debugging Tool.** We introduce a novel debugging tool that enables dedicated tracing and manipulation of LLM agents. This tool allows users to observe and intervene at arbitrary agent steps, facilitating efficient backtracking and debugging of complex multi-step processes.
- **LLM-Aided Debugger.** We develop an LLM-aided debugger that leverages self-reflection to identify and fix incorrect steps. This feature enhances debugging by automating the pinpointing and correction of errors, keeping the human in the loop to mitigate further hallucination.
- **Use Cases.** We demonstrate the utility of our tool by showcasing three practical use cases involving modern agents for data-driven applications: tracing data provenance; personalized recommendation; and debugging data quality. These illustrate the debugger's versatility in supporting varied and complex agent tasks.

2 SYSTEM DESCRIPTION

To facilitate interactive debugging, LADYBUG allows users to step through and intervene on the state of an agent in a manner similar to software debuggers. Software debuggers often facilitate *real-time* intervention by allowing users to suspend execution of the software program, interpret and modify its current state, then resume execution. We adopt an alternative *post-hoc* strategy that allows users to review the evolution of an agent's state, enqueue state modifications, then re-execute only the affected operations. In this sense, our debugger works in a *declarative* fashion, requiring users to declare all modifications upfront, while software debuggers work in an *imperative* fashion, requiring users to apply each modification individually in real-time. This declarative approach is better suited for the potentially lengthy traces of LLM agents. We additionally propose an LLM-aided debugger, to further support debugging at interactive speeds.

2.1 Preliminaries

We focus on an emerging class of LLM agents that model the completion of tasks dynamically in a declarative, event-driven *workflow* (e.g., workflows in LlamaIndex, flows in CrewAI, and state graphs in LangGraph). Let an agent A be defined by a set of *step functions* $F = \{f_1, f_2, \dots, f_k\}$, the atomic units of computation in an agent. A step function is invoked by some *input event*, executes some predefined instructions (i.e., a program), then returns some *output event*. We assume that step functions are invoked *serially*, such that each output event *triggers*—is passed as an input event to—one or more step functions. We assume that these events constitute snapshots of the agent's entire state; we do not model any other state or memory constructs that agents may choose to maintain.

Agents follow some internal policy (i.e., *control flow*) to coordinate invocation of step functions in F. We do not explicitly model this policy, but note that nondeterministic control flow could confound the debugging process. An agent invokes step functions in some order according to its control-flow policy, beginning with a predefined f_{start} and ending with some predefined f_{end}. We define a *trace* $T = (e_1, e_2, \dots, e_n)$ as a sequence of n step-function invocations in the order they were performed by A. Each $e_i = (p_i, f_i, o_i)$ records the output event o_i returned by step function f_i invoked with input event p_i, as dictated by $o_i = f_i(p_i)$.

2.2 Agent Debugger

In our demo, we present a full-stack software tool for debugging LLM agents, written in Python and TypeScript. LADYBUG allows users to run and debug LLM agents built with the LlamaIndex Python framework. Both LlamaIndex agent abstractions are supported: *FunctionCallingAgents* and *Workflows*. Agent and instrumentation-related capabilities are facilitated via the LlamaIndex framework, and all agents use the gpt-4o LLM via the OpenAI API. In concept, the LADYBUG *framework* is compatible with many event-driven agent frameworks; for the purposes of this demonstration, however, our *debugging tool* only supports LlamaIndex agents.

LADYBUG consists of three main components, as seen in Figure 2: a *database* (callout C); a *backend* (B); and a *frontend* (A).

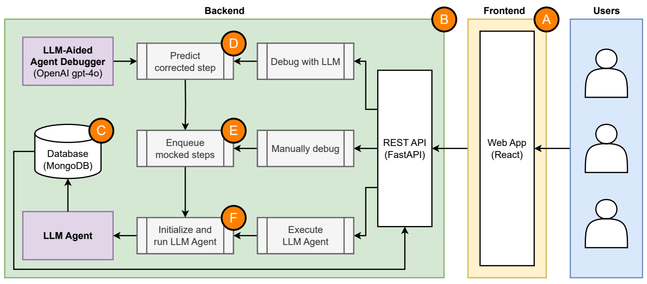


Figure 2: The architecture of LADYBUG.

The database is a local MongoDB instance, used by the backend to persist instrumentation data. The backend is a FastAPI server, which hosts the user's agent and defines API endpoints to facilitate debugging. The frontend is a React web application built with the React Material UI library. Users interact with this frontend to define inputs, run the agent, observe its trace, apply changes, and re-execute.

Assume that a user defines a LlamaIndex agent A, then launches our debugging tool for this agent. Assuming f_{start} is defined by the agent, the tool accepts a start input p_{start} from the user, triggers execution $f_{start}(p_{start})$, then awaits a final output o_n. LADYBUG hooks into A to record step-function invocations, which enables the construction of a completed trace T. The trace T is presented to the user in an interactive timeline view (callout #2 in Figure 1), allowing the user to inspect the chronology of step-function invocations and their input and output events (callout #5). By allowing users to run an agent, view outputs, and inspect internal state, our debugger facilitates *observability* for LLM agents.

To facilitate *debugging* of a completed trace T, users create a set of "mocked" invocations (*mocks*) $M_T = \{m_1, m_2, \ldots, m_j\}$ that preempt step-function invocations during re-execution (callouts E and F in Figure 2). Each mock $m_i = (p'_i, f'_i, o'_i)$ declares that, when re-executing A, any invocation of a step function f'_i with input event p'_i should simply return the mocked output event o'_i instead of actually invoking $f'_i(p'_i)$. This flexible design enables sophisticated interventions, where only certain invocations (i.e., a proper subset of T) are modified and re-executed, while supporting (and subsuming) the simpler case of a single intervention, where the intervention and all prior invocations would be included in M_T.

Since steps are the atomic units in our agent formulation, LADYBUG does not support intervention at any finer level of granularity. We argue that any instruction within a step can be easily mocked by decoupling it into a dedicated step, avoiding any need thereafter to recompile the agent. Ultimately, the user is free to choose which steps are mocked or re-executed, regardless of their position within the trace. This allows users to mock steps that are inconsequential or nondeterministic. We argue that this freedom is imperative when debugging LLM agents, as their control flow and steps (which will utilize nondeterministic LLMs) are not necessarily deterministic.

Beyond the ability to observe and debug LLM agent steps, LADYBUG tracks a variety of low-level events and metrics. We leverage the LlamaIndex instrumentation framework to monitor fine-grained statistics, namely system and user-defined *events* and *time spans*. Instrumentation events and spans are metadata objects corresponding to discrete moments and periods of time, respectively, and can be used to monitor various properties (e.g., frequency or length) of operations performed *within* agent steps (e.g., LLM calls). These primitives form a chronological hierarchy;

Instructions:
You are debugging a user's interaction with an AI agent, which completes a task given by the user. The interaction is denoted as a sequence of steps, each drawn from a list of valid step functions. Your task is to analyze the interaction and assess whether or not the output is correct. If it is incorrect, please trace the issue back to the first chronological incorrect step, and revise this step by proposing a corrected step function invocation. Output exactly 1 line, and absolutely nothing else. If you assess the output as being correct, simply output "N/A". Otherwise, output "Step N: C", where N is the step number of your revised step and C is the revised function invocation.

Valid Step Functions:
- preprocess_essay(ev: StartEvent) → RetrievalEvent
- retrieve_similar_essays(ev: RetrievalEvent) → SimilaritiesEvent
- identify_similarities(ev: SimilaritiesEvent) → PlagiarismEvent
- compute_plagiarism(ev: PlagiarismEvent) → SummarizeEvent
- summarize_plagiarism(ev: SummarizeEvent) → StopEvent

User Interaction:
- Input: StartEvent(essay=Essay(author="...", content="..."))
- Step 1: preprocess_essay(ev=StartEvent(...))
- Step 2: retrieve_similar_essays(ev=RetrievalEvent(...))
- Step 3: identify_similarities(ev=SimilaritiesEvent(...))
- Step 4: compute_plagiarism(ev=PlagiarismEvent(...))
- Step 5: summarize_plagiarism(ev=SummarizeEvent(...))
- Output: StopEvent(result="The probability of plagiarism is 82%")

Figure 3: The prompt used by the LLM-Aided Agent Debugger.

each event belongs to zero or one span, and each span has zero or one parent spans. Our tool records and persists any instrumentation events and spans emitted during execution, which are then extracted and reorganized into an interactive hierarchical timeline.

2.3 LLM-Aided Agent Debugging

When using LLM agents, users may encounter difficult debugging situations. For example, agents may produce lengthy traces, making the process of identifying mistakes difficult, and re-executing expensive. To help in these situations, LADYBUG offers an automated debugger aided by an LLM, in lieu of a user applying their own intuition. Finding the *first* anomaly in any data pipeline is critical to ensure that bad data is not passed downstream [3]. Therefore, we leverage the LLM to pinpoint the *first* incorrect invocation and predict a modified invocation that *corrects* the incorrect output o_n. This strategy exploits LLMs' surprising and well-documented ability to reflect upon previous responses and suggest corrections [5]. Our use of a separate LLM avoids any obligation for *self*-explanation. This modular design allows engineers to use a relatively expensive (i.e., more capable) LLM to debug agents that use cheaper LLMs.

To facilitate this strategy, we design a custom LLM prompt that describes the agent A, summarizes the trace T, and instructs the LLM to debug T. Figure 3 illustrates an example of such a prompt, tailored for debugging the plagiarism agent described in Section 3.1. In popular Python-based agent frameworks like LlamaIndex, step functions are defined as Python functions, therefore we can extract a variety of step function metadata (i.e., purpose, behavior, and possible return values) from the *signature* and *docstring* of these Python functions. We append the signature and docstring for each step function $f \in F$ to the prompt, then append each step-function invocation $e \in T$ in chronological order. When asking the LLM to debug T, we instruct it to first decide if the agent's output is incorrect, and if so, to trace the issue back to the first incorrect invocation (suppose this is at index k in T) and propose a *correct* invocation e'_k. A set of mocks $M_T = \{e_1, \ldots, e_{k-1}, e'_k\}$

is constructed (callout E in Figure 2) by appending e'_k (callout D) to the set of invocations preceding it in T, then re-execution commences in the same fashion as the manual debugging method (callout F).

3 DEMONSTRATION PLAN

Conference participants will explore strategies for tracing and debugging agent responses. They will then use these strategies to explore their own corrections and those suggested by LADYBUG.

3.1 Use Case #1: *Tracing Data Provenance*

The user has built an agent to detect and measure plagiarism in student essays by leveraging information retrieval (IR) techniques. The agent defines several step functions to complete this task, including a step for retrieving similar essays from a database of previous submissions, a step for identifying textual similarities between a pair of essays, and a final step that computes a plagiarism score based on these findings. When the user submits an input essay to the agent, the agent returns a plagiarism likelihood score of 82%, which appears unexpectedly high, given the essay's seemingly original content. The user seeks more context to understand how this conclusion was reached, and employs LADYBUG to trace the agent's sequence of operations and inspect intermediate outputs.

Using the execution trace provided by LADYBUG, the user verifies that the first step successfully retrieved similar essays, confirming that the retrieval techniques worked as expected. Reviewing the content of the retrieved essays, the user notices no glaring signs of plagiarism, prompting further investigation. Inspecting the output of the similarity-identification step, the user discovers that several inconspicuous sentences from one of the retrieved essays are repeated word-for-word in the input essay. Upon examining the metadata of the retrieved essays, the user finds that the matching essay was submitted by a person sharing the same family name as the input essay author, raising concerns about academic dishonesty. By exposing a detailed execution trace, LADYBUG has enabled the user to validate IR processes and trace data provenance in the agent, thereby instilling confidence in the original plagiarism score.

3.2 Use Case #2: *Personalized Recommendation*

The user is building an agent to assist students in exploring potential universities, by offering personalized recommendations based on diverse criteria. The agent defines a series of steps that mirror common IR techniques used in personalized search and recommendation systems. These steps include identifying prospective universities, several costly steps to scrape web data and commentary about each university, a step to generate ranking criteria based on student preferences, a step to rank universities according to these criteria, and a final step to synthesize a personalized summary of the best-matched university. The user initially queries the agent for recommendations for a computer-science degree, emphasizing criteria related to industry and research impact. After executing, the user receives a recommendation for the University of Toronto, which is justified by the university's reputation for producing high-quality AI research.

This recommendation aligns with the user's preferences; however, the user questions why the University of Waterloo, known for its strong co-operative education program and industrial connections, was not recommended. Through the trace provided by LADYBUG, the user examines the criteria generated by the agent, which emphasized traits such as "number of research publications" and "industry impact." Curious to explore a different outcome, the user leverages the interactive debugging capabilities of LADYBUG to adjust the criteria without re-running the expensive web scraping steps. By selectively mocking the criteria to include "industry connections" and "employment opportunities," the user re-executes the agent and discovers that it now recommends the University of Waterloo. This exploration helps the user understand the agent's recommendations, and shows how interactive debugging can facilitate personalization and query expansion.

3.3 Use Case #3: *Debugging Data Quality*

As introduced in Section 1, suppose the user is building an agent that grades student essays, coordinating various steps to assess and process student submissions. The user loads the agent into LADYBUG and runs it on a large collection of student essays. Upon completion, the agent produces a grade distribution that reports an impossible class average of 1222.92% (callout #1 in Figure 1). In LADYBUG, the user begins by reviewing the initial "setup" step, and finds that the rubric's grading criteria weighting is *non-uniform*, which may have skewed the average. To verify this hypothesis, the user creates a mock for this step, modifying the output to prescribe *uniform* weighting. Upon re-executing, the user finds that the class average reduced slightly; however, it is still anomalous.

After thorough review, the user is unable to identify any further issues (#2). Due to the lengthy nature of the execution trace, however, the user suspects that there may be inconspicuous data quality or control flow issues. The user turns to the LLM-aided debugger (#3), which identifies an inconsistency in the lengthy trace: in a late "assess quality" step (#4), the LLM inadvertently hallucinated *integer* rubric scores instead of *float* scores, which skewed distribution calculations in subsequent steps. As shown in Figure 1, the debugger suggests a revision to this "assess quality" step that replaces the integer scores with their corresponding float scores (#5). The debugger executes this revision, which re-executes only the five steps that follow the revised step, and displays the new average of 67.92% in the interface (#6). Through self-reflection, the LLM-based debugger has enabled the user to identify, trace, and correct data quality issues within this complex agent workflow.

REFERENCES

[1] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *PVLDB* 17, 12 (Nov. 2024), 4213–4216.

[2] El Kindi Rezig, Ashrita Brahmaroutu, Nesime Tatbul, Mourad Ouzzani, Nan Tang, Timothy Mattson, Samuel Madden, and Michael Stonebraker. 2020. Debugging large-scale data science pipelines using dagger. *PVLDB* 13, 12 (Aug. 2020), 2993–2996.

[3] El Kindi Rezig, Lei Cao, Michael Stonebraker, Giovanni Simonini, Wenbo Tao, Samuel Madden, Mourad Ouzzani, Nan Tang, and Ahmed K. Elmagarmid. 2019. Data Civilizer 2.0: a holistic framework for data preparation and analytics. *PVLDB* 12, 12 (Aug. 2019), 1954–1957.

[4] Joel Rorseth, Parke Godfrey, Lukasz Golab, Divesh Srivastava, and Jaroslaw Szlichta. 2024. RAGE Against the Machine: Retrieval-Augmented LLM Explanations. In *ICDE*. 5469–5472.

[5] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *NeurIPS* 36 (2024).

[6] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*.