Watson: A Cognitive Observability Framework for the Reasoning of Foundation Model-Powered Agents

Benjamin Rombaut*, Sogol Masoumzadeh†*, Kirill Vasilevski*, Dayi Lin*, Ahmed E. Hassan‡
*Centre for Software Excellence, Huawei Canada
†McGill University, Montréal, Canada
‡Queen's University, Kingston, Canada

Abstract—As foundation models (FMs) play an increasingly prominent role in complex software systems, such as FM-powered agentic software (i.e., Agentware), they introduce significant challenges for developers regarding observability. Unlike traditional software, agents operate autonomously, using extensive data and opaque implicit reasoning, making it difficult to observe and understand their behavior during runtime, especially when they take unexpected actions or encounter errors. In this paper, we highlight the limitations of traditional operational observability in the context of FM-powered software, and introduce cognitive observability as a new type of required observability that has emerged for such innovative systems. We then propose a novel framework that provides cognitive observability into the implicit reasoning processes of agents (a.k.a. reasoning observability), and demonstrate the effectiveness of our framework in boosting the debuggability of Agentware and, in turn, the abilities of an Agentware through a case study on AutoCodeRover, a cutting-edge Agentware for autonomous program improvement.

Index Terms—Observability, Foundation models, Large Language Models, AIware, FMware

I. INTRODUCTION

Agentic software (i.e., Agentware [23]), powered by foundation models (FMs) with reasoning capabilities [6, 28, 46], are gaining significant traction in a variety of domains such as autonomous software development [31, 74], customer support [20], and data analytics [19]. However, such agentic software present new challenges compared to traditional software [24], especially in the realm of observability, as they operate with high levels of autonomy and unpredictability, and make decisions through implicit reasoning that is often opaque [68]. This opacity complicates the ability to monitor, debug, and ensure the reliability of such software. For instance, an FM-powered applicant recommendation system has been found to manifest biases towards certain applicants based on their specific traits [11]. Despite significant efforts, engineers are often not able to identify the root causes of these inherited biases due to the "black-box" nature of the reasoning behind such agentic software.

Observability techniques applied to traditional software typically involve principles related to *operational observability*—monitoring metrics and system status through logs, traces, and performance counters [40, 57]. Such methods are effective due to the direct traceability between the system's deterministic behavior and the development assets that are being executed, such as code snippets and predefined workflows. Developers can easily instrument such development assets (e.g., code) to track execution flows, identify bottlenecks, and detect anomalies. Issues can be traced back to and addressed at the respective assets.

However, while operational observability has also shown its value for developers and operators of Agentware by tracing model inference calls, tool usage, token consumption, and intermediate outputs at a system level, it falls short of providing adequate insights for diagnosing, debugging, and addressing issues in the implicit reasoning process of agents. Agents often operate with a high degree of independence, making decisions based on evolving and dynamically retrieved contexts, learned experiences, and interactions with other agents or environments. The underlying logic driving agent behavior is not traceable through conventional logs or performance counters, as the behavior of agents is not specified in code, but rather instructed through prompts and reasoned and planned by FMs that have extremely limited inherent explainability. As prompts and FMs are not instrumentable and do not produce deterministic or reproducible execution results, it is extremely challenging to pinpoint failures or understand why an agent behaves in a certain way. As such, new observability concepts and approaches are required that are capable of capturing not just system outputs, but also the cognitive-like reasoning processes of agents.

For this purpose, we propose a new type of observability called *cognitive observability* for Agentware, which enables the rationale of agent actions and decisions within a broader environment to be observed, providing insight not only into what agents do, but also why they do it. This type of observability is crucial for debugging agentic software, as it helps track reasoning pathways, evaluate how knowledge and contexts influence agent behaviors, and optimize the alignment between agent decisions and desired outcomes.

The need for cognitive observability becomes more prominent with the latest Agentware, which adopts enhanced reasoning capabilities provided by proprietary novel reasoning models that do not reveal their implicit reasoning paths to agent developers. For example, OpenAI's o1 model series [32] are designed to spend more time reasoning before they respond, which allows them to articulate their reasoning processes more

clearly while working through complex tasks [5, 64, 69, 72, 73]. However, despite these improvements, the developers of o1 opted not to reveal its reasoning process due to policy compliance or competitive advantage [38]. Additionally, it is not always possible to include the reasoning in the response of reasoning models, as downstream systems might be tightly coupled to the model outputs and require a strictly defined structured output format from such models.

Furthermore, in the context of Agentware, effective observability is not only essential for human operators to understand how these systems function, but also for the systems themselves. Because Agentware systems include both human operators and autonomous agents, they can leverage observability techniques to enhance their system-understanding by providing insights into the internal states and processes of the whole software and even themselves. This capability for system-awareness means that effective observability enables Agentware to adapt and improve over time, which is critical in developing more capable and resilient systems. In fact, agents debating and reflecting on each other's decisions has been shown to significantly improve system performance [67], and by allowing one agent's reasoning process to be observed by other agents, such cross-checking has the potential to further boost system performance.

In this paper, we extend the current observability taxonomy of Agentware by integrating the concepts around cognitive observability. We then introduce a novel framework, called Watson, to observe the reasoning path of agents, a primary consideration of cognitive observability. We demonstrate the effectiveness of Watson on AutoCodeRover [74], a state-of-the-art Agentware for autonomous program improvement. Our case study shows that Watson enables developers to identify defective reasonings in agents, which in turn can help optimize the capabilities of the Agentware, by debugging and addressing two failed instances of AutoCodeRover on SWE-bench-lite [58]. The main contributions of this paper are as follows:

- We propose a taxonomy of observability for Agentware, highlighting the unique challenges and opportunities, with a focus on the limitations of operational observability and the need for cognitive (in particular reasoning) observability.
- We propose a novel framework, Watson, for observing an agent's implicit reasoning processes, contributing to more effective monitoring and debugging of Agentware.
- We demonstrate the effectiveness of our framework on a state-of-the-art Agentware, AutoCodeRover, showcasing the effectiveness of cognitive observability in improving agent capabilities and debuggability.

The remainder of the paper is structured as follows: Section II discusses observability for Agentware. Section III introduces Watson, our proposed framework. Section IV demonstrates Watson on AutoCodeRover. Section V examines important points from our case study. Section VI discusses the threats to the validity of our study. Finally, Section VII concludes the paper.

II. OBSERVABILITY FOR AGENTWARE

In this section, we first introduce our proposed extended taxonomy for the observability of Agentware. We then discuss related work in operational (Section II-B) and cognitive (Section II-C) observability, as well as highlight the challenges of both types of observability as applied to Agentware, and FMware in general. Our taxonomy and their associated challenges were identified based on not only in-depth discussions with industrial and academic leaders (e.g., during *SEMLA 2023* and *2024* [35], *FM+SE Vision 2030* [22], *FM+SE Summit 2024* [21], and *SE 2030 workshop—FSE 2024* [52, 53] events), but also our practical experience with the research and development of FMware [23, 41], and our close interactions with over 40 leading industrial partners (e.g., Intel, AMD, RedHat, HuggingFace, and SAP) as part of the Open Platform for Enterprise AI (OPEA) alliance [48].

A. Taxonomy of FMware Observability

Agentware observability (Table I) encompasses two key categories: operational observability and cognitive observability, which together provide a comprehensive view of software behavior and insights. Operational observability focuses on monitoring software performance and resource consumption, with sub-categories like logs & program execution traces (e.g., capturing detailed events), volume metrics (e.g., measuring data or task throughput), latency metrics (e.g., tracking response times), token metrics (e.g., monitoring token usage in models), resource utilization (e.g., evaluating cost and computational resource consumption), and FM tool instrumentation (e.g., gathering metrics from FM tools such as document retrievers [43]). Cognitive observability refers to understanding the behavior and quality of Agentware at a cognitive level that often cannot be quantified or monitored easily by operational observability, such as reasoning paths (i.e., understanding the implicit reasoning path of an agent), semantic feedback (e.g., gathering explicit and implicit user responses to the system output, such as modifications to the output), and output integrity (e.g., analyzing model outputs for qualitative insights and hallucinations [29]).

B. Operational Observability

Operational observability remains crucial for any FMware, including Agentware, just as it is for traditional software [40]. It provides vital insights into software performance and behavior, ensuring that potential issues can be identified and addressed before they impact users. This is especially important when FMware involves complex workflows that integrate multiple model inferences and legacy system calls, as operational observability enables a comprehensive view of inter-system and inter-component interactions.

Even when treating the FM as a "black box", we can still gain observability into their behaviors by analyzing logs and execution traces of FMware, which reveal data flow

TABLE I: The taxonomy of observability for Agentware.

Category	Sub-Category	Description
Operational Observability	Logs & Execution Traces	Capturing detailed system events
	Volume Metrics	Measuring data or task throughput
	Latency Metrics	Tracking response times
	Token Metrics	Monitoring token usage in models
	Resource Utilization	Evaluating computational resource consumption
	FM Tool Instrumentation	Gathering metrics from large language model tools
Cognitive Observability	Reasoning Path	Tracking model reasoning processes
	Semantic Feedback	Gathering explicit and implicit user responses
	Output Integrity	Analyzing FM outputs for qualitative insights

Fig. 1: In multi-agent systems, errors made by early agents may only manifest later in the workflow.

between components and help detect anomalies in execution paths [9]. Volume metrics offer a way to track the number of calls to specific FMs, while latency metrics measure delays in responses across FMware, helping to identify potential bottlenecks and improve the user experience. Token metrics provide insight into model behavior, revealing patterns of prompt and completion usage that can impact overall performance and cost, while tracking resource utilization ensures that compute, memory, and storage capacities are used by FMs efficiently [1]. Additionally, monitoring the tools that FMs use is crucial for understanding how they interact with other FMware components, as this enables us to assess their capabilities and effectiveness in the overall architecture of the FMware. By gaining observability into these categories, we can optimize the capabilities of FMware, enhance collaboration among FMware components, and ensure that the FMware operates smoothly and efficiently, ultimately leading to a better experience for all FMware users.

State-of-the-art FM observability platforms like Lang-Smith [37], TraceLoop [60], Weights & Biases [65], and others [15, 25, 30, 36, 45, 54, 55, 66] have already begun to offer support for tracking operational observability metrics for FMware. For example, the open-source OpenLLMetry [61] package from TraceLoop is designed to offer standard Open-Telemetry [49] instrumentation for FM providers and vector databases. This enables developers to gain basic observability over their FMware, allowing them to monitor prompts, token usage, and grounding accuracy over time. While these products provide valuable insights into operational metrics, they often overlook the contextual factors and implicit reasoning paths of FMware, falling short of offering adequate insights to help developers assess the user-perceived quality and debug issues within reasoning paths.

C. Cognitive Observability

The key to truly enhancing observability for FMware lies in implementing cognitive observability, which shifts the focus to higher-level cognitive and linguistic aspects. To address these comp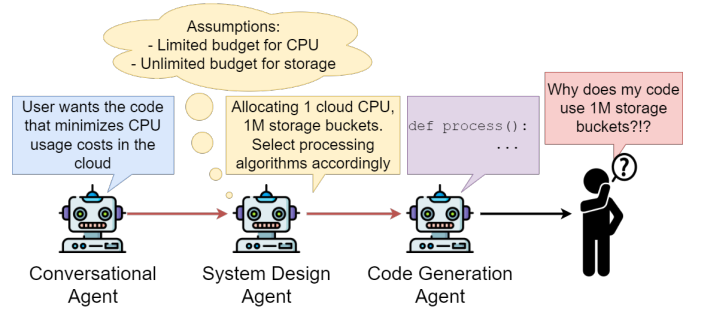lexities, cognitive observability focuses on three critical sub-categories: reasoning path, semantic feedback, and output integrity. Reasoning path tracks the reasoning process of FMs, offering insights into how they generate conclusions based on their often opaque decisions. Semantic feedback gathers and interprets user feedback on the FMware to improve system behavior. Finally, output integrity analyzes model outputs to derive qualitative insights and identify and mitigate malicious prompts, sensitive data, toxic responses, problematic topics, and hallucinations. These categories provide a comprehensive view of FMware performance, essential for managing complex, multi-agent systems and addressing challenges that traditional observability alone cannot solve.

Reasoning Path: This sub-category focuses on tracking FM reasoning paths, but faces several challenges. One significant challenge is the need for higher-level abstractions that go beyond simply logging inputs and outputs. Understanding how models arrive at specific conclusions requires insight into their implicit reasoning paths, which often involve complex algorithms, knowledge selection, and decision-making processes that are not easily interpretable. For example, in the case of retrieval-augmented generation (RAG) [39], although we can evaluate the retriever's performance independent of the rest of FMware through operational observability (i.e., monitoring the component output), there is no guarantee that the retrieved context is used correctly, or even used at all, during the reasoning process. Knowing this is essential for debugging decisions made by agents, particularly in complex FMware where potentially many agents interact and coordinate with both legacy systems and other agents. Without this higher-level abstraction, it becomes difficult to pinpoint why an agent took a specific action or made a particular decision, and understanding the reasoning process or inference chain that led to an erroneous decision is crucial for helping FMware developers understand how to correct potential errors. Still, current tools offer limited visibility into these reasoning pathways, which become even more important in observing as FMware agent reasoning capabilities improve [56].

However, we cannot simply instruct agents in FMware to output their reasoning, because doing so inherently alters the completion [64]. This is similar to the observer effect in quantum physics, where observing a system changes its state [12].

When an agent is asked to explain its reasoning, it is no longer just completing its original task; it now has to simultaneously generate an explanation, which can shift its focus or introduce new biases into its responses. This alteration complicates the ability to assess the agent's original reasoning process in a pure, unmodified form, making it challenging to diagnose issues or understand the unobserved logic behind its actions. Additionally, integrating reasoning into the output of FMs may not always be feasible, as downstream systems in FMware can be tightly coupled to the FM's outputs and may demand a strictly defined and structured output format, limiting the agent's ability to provide comprehensive explanations without compromising its primary functions.

When errors in FMware occur, it is not trivial to localize where the fault originated in the system. In traditional software systems, logs and stack traces provide clear insights into the source of an issue, allowing developers to trace errors to specific lines of code or API calls. However, in FMware, errors often arise from more abstract or emergent behaviors that are not easily observable because there is no explicit code driving the behavior of FM agents. This lack of traditional code structure means that errors can't always be pinpointed through conventional debugging techniques, further complicating understanding why agents behave the way they do. These errors are also harder to trace because they don't always stem from a single point of failure but rather from complex reasoning chains or interactions across FMware. There is no clear logic to inspect or trace because much of the decision-making is based on latent knowledge embedded within the model. Additionally, in multi-agent FMware systems, errors made by early agents in FMware systems may only manifest later in the workflow, further complicating the ability to trace back the source of a problem. For instance, an incorrect assumption or action by one agent can lead to a cascade of misaligned actions by downstream agents, resulting in unpredictable outcomes, as demonstrated in Figure 1.

Previous attempts have been suggested to observe the reasoning paths of agents in FMware, in which the cause-and-effect actions (i.e., when x happens, then y follows) of agents are tracked as a proxy to show their implicit reasoning paths [27, 44, 51]. Additionally, the benefits of enabling "self-explainability" in FMware agents, for the purpose of enhancing their observability [26], transparency [63, 71], and building trust with end-users [50, 76], have also been investigated. Zelikman et al. [70] are particularly relevant as they focus on generating reasonings that guide the FM towards the correct answer by reasoning backward from the solution, ultimately enhancing the model's accuracy. Our focus on observing reasoning paths diverges significantly in two key areas. First, our goal is not to ensure the correctness of the answer, but rather to understand how agents reason, irrespective of whether their completion is correct. Second, the methodologies differ: since their work only aims to generate reasoning that leads to the correct answer, it is not necessary to verify whether the generated reasoning aligns with the model's actual reasoning process. In contrast, our approach introduces verification

mechanisms, including `EOM` token verification (Section III-A) and prompt attribution validation (Section IV-B), to ensure that the generated reasoning reflects the model's true reasoning process.

Semantic Feedback: The tracking of cognitive observability includes another crucial component: capturing semantic feedback in the form of explicit, implicit, and free-form feedback from users and systems [47]. Explicit feedback includes direct actions on rating FMware-generated content. For example, OpenAI's ChatGPT [7], as well as existing FMware observability tools [25, 30, 37, 54], allow users to assign a "thumbs up/down" rating to FM completions. Implicit feedback is drawn from user behaviors, such as whether the user "copied", "saved", or "dismissed" the completion. For example, GitHub Copilot [17] checks whether a code suggestion that was accepted by the developer is still present in the code at different time intervals after the insertion, as well as the extent to which the user edited the code suggestion [10], to evaluate the utility of the suggestion. Free-form feedback consists of corrections or explanations provided by users regarding the generated output of FMware [30]. These three types of feedback metrics help developers gauge the efficacy of FMware output, optimize the user experience of FMware agentic applications, and identify trends that inform decisions about model training or fine-tuning.

However, gathering semantic feedback presents its own challenges, particularly in the context of the limited observability of FMware. While user interactions provide valuable explicit and implicit responses, interpreting these signals can become complex as interactions in FMware grow more intricate, especially in multi-agent systems. Agents may dynamically adapt their responses based on not only their learning but also the interactions of other agents in the system, and this rising complexity can make it difficult to pinpoint specific areas for improvement or obscure the relationship between user feedback and agent behavior.

Output Integrity: The focus of output integrity is on analyzing the output of FM agents to gain qualitative insights into their behavior. This involves techniques like sentiment analysis to understand emotional tone [45, 66], as well as detecting hallucinations——instances where the model generates information that is factually incorrect or irrelevant [29]. These signals provide critical feedback on the reliability and coherence of the model's output, helping to identify inconsistencies, biases, or errors that may not be evident from traditional metrics alone.

Analyzing FM outputs for qualitative insights also comes with challenges, particularly in environments with rising complexity and dynamic interactions. As FMs commonly generate text outputs, understanding the nuances in these outputs requires sophisticated methods to assess their relevance and coherence. The dynamic nature of multi-agent systems can further complicate this analysis, as the context in which outputs are generated may change rapidly due to interactions with other agents, which can lead to situations where outputs
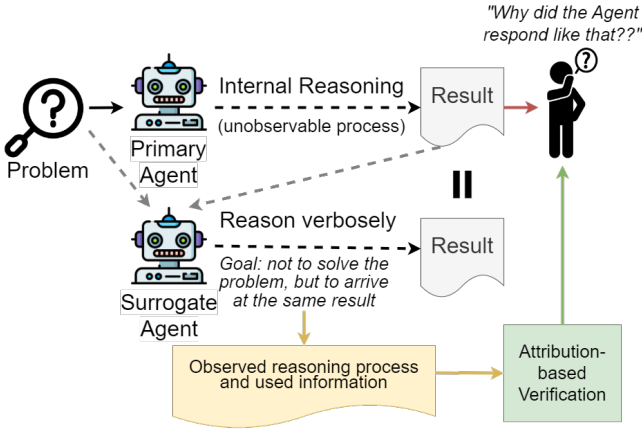
Fig. 2: Overview of Watson—a framework to observe the reasoning process of agents without affecting their behavior.

are difficult to evaluate effectively, limiting observability and insight. Additionally, the interplay between language generation and reasoning paths complicates deriving actionable insights from output integrity, as the reasons behind certain outputs may not be immediately apparent.

III. WATSON: A REASONING PATH FRAMEWORK

To address the aforementioned challenges, specifically those related to reasoning path, we introduce Watson, a framework to observe the reasoning process of agents without affecting their behavior (Figure 2). In this framework, a "surrogate agent" [42] operates in parallel with an original agent of interest (a.k.a the "primary agent") for observing its implicit reasoning process. The goal of the surrogate agent is not to arrive at a "correct" completion, but rather to replicate the primary agent's completion, while also reasoning verbosely about its thought process for arriving at that outcome. As such, we can observe the implicit reasoning path used by the primary agent without affecting the original completion as a result of asking the agent to output its reasoning process. Agentware developers benefit from this reasoning, gaining insight into why the primary agent responded as it did and using the reasoning as "hints" to debug or enhance the performance of the primary agent without altering its natural behavior.

A. Overview of Watson

Watson is structured around three primary considerations. First, the surrogate agent must mirror the configuration of the primary agent, allowing for the replication of the primary agent's behavior and ensuring similar performance among the two agents. Next, Watson employs a mechanism for generating reasoning. Finally, Watson ensures the generated reasonings are aligned with the targeted completion. We explain each consideration in detail below.

Mirroring Configuration of Primary Agent: To ensure the surrogate agent precisely replicates the primary agent's implicit reasoning process, it must maintain a mirrored configuration, including the FM architecture and key decoding

parameters. By using the same FM, the surrogate agent interprets the inputs with the same underlying capabilities as the primary agent. Furthermore, decoding parameters, such as *temperature* and *top_p*, must be identical between the two agents, as these parameters govern the determinism and creativity of the model's outputs, where even slight variations could lead to divergence in agent behavior. It is worth noting that identical configurations do not guarantee identical or reproducible model output [8]. We address the concern in the final consideration of Watson.

Generating Reasoning Paths: With the goal of Watson being to generate an implicit reasoning path that ties an input prompt to the completion of a primary agent, the surrogate agent must access the primary agent's input prompt as the *prefix* and the primary agent's completion as the *suffix*, and generate completion tokens to bridge their gap with a reasoning path. However, this poses a novel challenge for designing Watson, as autoregressive FMs like the popular OpenAI's GPT [2] and Meta's LLaMa [59] are built using causal decoder-only transformer architectures [62] and are trained to predict the next token in a left-to-right fashion. In other words, the generation of the next completion token only depends on the given tokens in the input prompt and the already-generated tokens in the completion.

To solve this problem, a *fill-in-the-middle* (FIM) technique [4] is proposed for enabling causal decoder-based FMs to use both the prefix and suffix to infill a middle region of a prompt, primarily motivated for code completion applications [16, 33]. The FIM approach (Figure 3) divides a prompt into a prefix, middle, and suffix identified with <PRE>, <MID> - <EOM> (i.e., start-of-middle and end-of-middle), and <SUF> special tokens, respectively. During training, the FM uses the prefix and suffix content as the input prompt, and attempts to generate the content in the middle.

To implement FIM, we arrange the input prompt as *prefix-suffix-middle* [4, 18], meaning the surrogate agent uses the primary agent's prompt as the prefix and the primary agent's completion as the suffix, followed by the <MID> token. The surrogate agent then generates the reasoning, with the <EOM> token marking its end. To ensure the proper conduct of the FIM task, we explicitly check for the generation of the <EOM> token. As such, the generated reasoning by the surrogate agent is a verified implicit reasoning path for linking the primary agent's input prompt and its corresponding completion.

While FIM is ideal for our goal of generating reasoning paths, the approach is constrained to FMs that inherently support the FIM technique and can generate infilled completions. Consequently, it would prove infeasible for the surrogate agent to use the FIM technique if the primary agent does not inherently support the FIM technique. In such cases, we propose an alternative technique named "Repetitive Chain-of-Thought" for reasoning generation, which is not specific to FMs with FIM capability. This technique involves prompting the surrogate agent with the same input as the primary agent but adds an instruction to explicitly generate chain-of-thought

The early bird catches the worm, but the second mouse gets the cheese.

<PRE> The early bird catches the worm, but <MID> the second mouse <EOM><SUF> gets the cheese.

<PRE> The early bird catches the worm, but <SUF> gets the cheese. <MID> the second mouse <EOM>

Fig. 3: Example of FIM prompt with the special tokens; top - original prompt, middle - original prompt with special tokens inserted; bottom - final prompt in prefix-suffix-middle mode.

reasoning [64] in its completion. To ensure alignment, any reasoning generated by the surrogate agent leading to an answer different from the primary agent's final completion is discarded. This alignment step is unnecessary in the FIM technique as the surrogate agent's completion is guided by the primary agent's output, which is provided as the suffix of the prompt. However, this step is crucial in Repetitive Chain-of-Thought as it guarantees that the surrogate agent's reasoning remains consistent with the primary agent's implicit reasoning process, ensuring that we capture only the most relevant reasoning paths. This filtering process minimizes discrepancies and ensures that the reasoning we analyze is closely tied to the primary agent's behavior. Using this approach, it is possible to capture the reasoning paths of the surrogate agent, and consequently the implicit reasoning of the primary agent, without the necessity for their underlying FMs supporting FIM. However, depending on the rounds of inferences that need to be filtered out due to inconsistent answers, such a technique may result in higher inference costs.

Verification of the Reasoning Consistency: It is necessary to ensure that the reasoning generated by the surrogate agent is aligned with the primary agent's completion so that the surrogate's reasoning reflects the implicit reasoning process of the primary agent. For this purpose, we task the surrogate agent with generating multiple potential reasoning paths so that we can extract the mutual threads and recurring ideas [72]. Doing so also allows for capturing different angles from which the primary agent might have addressed the input prompt, providing a more comprehensive view of its implicit reasoning process.

After extracting the consistent threads from the generated reasonings, we validate their alignment with the input prompt using PromptExp [14], a cross-granularity prompt explanation technique that calculates the importance of different components in the input prompt (a.k.a its attribution) in influencing the agent's completion. In other words, we confirm the ideas represented by the components of the prompt with the highest attributions are emphasized within the generated reasoning. For calculating these attributions, we tokenize the input prompt and create a perturbed version of the input by removing one token at a time, effectively masking it [75]. This perturbed prompt is then passed to the primary agent for a new completion. To evaluate the importance of the masked input token, we calculate probability differences among analogous tokens of the new completion and the completion corresponding to the original input prompt. A higher arithmetic mean of these differences results in a higher attribution value for the masked

input token, which in turn indicates its greater importance for the agent completion. We then aggregate the token attributions to different granularities of the prompt using the method proposed in PromptExp.

IV. DEBUGGING AUTOCODEROVER WITH WATSON

In this section, we conduct a case study of using Watson for debugging AutoCodeRover, a state-of-the-art Agentware for autonomous program improvement, with 30.67% efficacy (pass@1) [3] on SWE-bench-lite, a subset of 300 issue instances from the SWE-bench [34] evaluation framework.

A. Motivation

AutoCodeRover operates in two phases. In the initial phase, an agent is tasked with analyzing an issue report and identifying corresponding defective code locations in the associated software project. The locations are identified by leveraging a predefined set of APIs that allows for the agent to examine the software project associated with the issue report and gather context of components in the project that might be relevant to resolving the issue. The retrieved context is then learned by the agent, refining its understanding of the code structure and issue report. If needed, the agent can re-invoke the same collection of API calls to gather additional context. This iterative process continues until the agent determines that it has sufficient context to identify the appropriate code locations for applying a solution patch, at which point the agent returns a *blank* completion in response to the input prompt. Once the defective code locations are decided, they are passed to a second agent, which is tasked with generating the solution patch. However, this is an opaque decision made by the first agent—it is not known why the agent decided it had enough context, or which parts of the already retrieved context it considers sufficient in making this decision. Therefore, the Agentware developers are not able to effectively debug the decisions of the agent.

B. Methodology

We apply Watson to AutoCodeRover to understand the implicit reasoning process leading to the agent's decision on the defective code locations. We use GPT-3.5-turbo-instruct as the underlying FM in AutoCodeRover (i.e., the primary agent) and in Watson (i.e., the surrogate agent), as this FM supports FIM. We select the following two SWE-bench-lite instances for which AutoCodeRover has made an attempt to solve. For both instances, incorrect code files were identified by the first agent as buggy locations, while the agent's reasoning for this decision is not immediately clear.

Listing 1: The generated reasoning from Watson for Instance `django__django-13401`

```
The common points across the provided explanations can be summarized as follows:
1. **Identified Bug Location**: All reasonings consistently highlight that the issue is centered around
    the model_inheritance/tests.py file , which contains classes A, B, and C. This file is repeatedly
    mentioned as critical to understanding where potential bugs may lie.
2. **Focus on Classes B and C**: There is a shared emphasis on examining classes B and C due to their
    inheritance from class A, suggesting that any issues likely originate from these subclasses or
    their relationship with the abstract base class.}@*)
3. **Method Investigations**: Several explanations point out specific methods ('__eq__', '__lt__', and
    '__hash__') in relation to fields within these classes, indicating a need to analyze how equality
    comparisons are implemented for instances of these models.
4. **Sufficient Context Established**: Multiple reasonings express confidence that enough context has
    been gathered regarding relevant files and classes without needing further information at this
    stage.
5. **Search Strategies Proposed**: Some responses suggest utilizing search functions like '
    search_method_in_class()' or looking for similar cases using other search tools to uncover
    additional insights about how fields are compared across different models.
Unique perspectives include:
- Explanation 6 specifically mentions adjusting methods related to Django's Field class.
- Explanation 7 provides quantitative data about matches found for each class in the codebase.
- Explanation 4 notes there could be multiple definitions of class names (e.g., two classes named B),
    hinting at possible name conflicts or confusion that might arise during debugging efforts.
Overall, while all explanations converge on key themes surrounding bug identification in a specific
    code file involving certain model classes, they also offer unique angles regarding method
    adjustments and contextual searches needed for thorough investigation.
```

Listing 2: A segment of the output from AutoCodeRover reapplied on `django__django-13401` with the reasoning hint

```
[ASSISTANT] The issue reported is related to the equality comparison of two fields in different models.
    The author found that if they pull the fields of two models into a shared set, one will be de-
    duplicated away, because they compare as equal. The root of the issue is that they compare equal,
    as fields only consider self.creation_counter when comparing for equality. So, we need to adjust
    __eq__ so that if the field.model is different, they will compare unequal. Similarly, it is
    probably wise to adjust __hash__ and __lt__ to match. When adjusting __lt__, it may be wise to
    order first by self.creation_counter so that cases not affected by this equality collision won't be
    re-ordered. In the experimental branch, there was one test that broke if ordered them by model
    first. The fix for this issue is not located in any tests.py files , although they are a good place
    to start to get more context on the issue. We also found that the code snippets we searched for
    are mostly located in the file django/db/models/fields/__init__.py , which makes sense as this is
    where the Field class is defined. We can also see that the class A is inherited by classes B and C,
    and B and C are inherited by other classes in the codebase. This could be the reason why the issue
    is not located in any 'tests.py' files, as it may be related to the inheritance of classes. Based
    on the context we have gathered, we can now start writing patches to solve the issue.
```

Instance django__django-13401[1]: This issue is reported in the Django [13] web framework and describes a problem where attribute fields of different models inherited from the same parent model are considered equal. This occurs because field comparison for equality is only based on `self.creation_counter`. As a result, attribute fields from models B and C, both inheriting from model A, are considered to be equal, even though their parent models are different, leading to unexpected behavior (e.g. if the same attribute fields from models B and C are added to a set, one of them will be deduplicated away). The author of the issue report suggests modifying the `__eq__`, `__hash__`, and `__lt__` methods to ensure fields with different parent models compare as unequal.

Instance django__django-14238[2]: This issue is reported in the Django framework and describes an application crash during model initialization when `DEFAULT_AUTO_FIELD` is set to a custom subclass of `BigAutoField`. The issue report includes a traceback demonstrating a `ValueError` requiring that the custom primary key class must subclass `AutoField`. The issue stems from a failure in `AutoFieldMeta.__subclasscheck__`, which does not recognize valid subclasses of `BigAutoField` or `SmallAutoField`. The issue report includes a suggested fix to modify the subclass check to include these subclasses in the `_subclasses` property.

After generating the reasonings using Watson, we use them to identify faulty decisions of the AutoCodeRover agent which lead it to report incorrect buggy locations. Using this

---

[1]https://code.djangoproject.com/ticket/31750

[2]https://code.djangoproject.com/ticket/32620

information, we provide a decision refining hint, alongside the initial input prompt, to AutoCodeRover for re-solving the aforementioned SWE-bench-lite instances, aiming for a correct solution with the help of the hints.

C. Results

In this section, we discuss the results of running Watson on the two instances described in Section IV-B. For each instance, we discuss the output of reasoning from Watson, the validation of this reasoning against the prompt attributions of the primary agent, and the outcome of re-running AutoCodeRover with corresponding hints to the aforementioned instances.

Instance `django__django-13401`

Output of Reasoning: The reasoning generated from Watson for `django__django-13401`, shown in Listing 1, suggests the agent is focusing on the test files, specifically `model_inheritance/tests.py`, which is not the code location for where we expect the solution patch to be applied. Following this incorrect decision pathway, AutoCodeRover attempts to modify subclasses `B` and `C`, which are contained in `model_inheritance/tests.py`, as the fix for the encountered issue.

Validation: With the inspection of PromptExp for the current instance shown in Figure 4, it becomes evident that, compared to the rest of the input prompt, `model_inheritance/tests.py` has a higher attribution value (Figure 4a). The same applies for subclasses `B` and `C`, confirming that the AutoCodeRover agent aims to write a solution patch by modifying `model_inheritance/tests.py` and consequently subclasses `B` and `C` (Figure 4b).

After Fixing with Hints: Based on the reasonings, we add the following hint at the end of the issue report and rerun AutoCodeRover.

> *The fix for this issue is not located in any "tests.py" files, although they are a good place to start to get more context on the issue.*

With this hint, AutoCodeRover arrives at the correct file location to be patched, while specifically acknowledging that the fix is not located in any test files (Listing 2).

Instance `django__django-14238`

Output of Reasoning: The reasoning generated from Watson for `django__django-14238`, shown in Figure 3, suggests that the agent is placing its focus on the files in the `example` directory. However, upon closer inspection of the original issue report, it is explicitly mentioned that the fix can be applied to the `__subclasscheck__` method, which does not show up at all in the reasonings, indicating that the agent is ignoring this information.

Validation: Upon closer inspection of PromptExp for the current instance, illustrated in Figure 5, it is evident that the attribution value of `__subclasscheck__` is lower compared to other components of the prompt—AutoCodeRover agent

does not consider the method `__subclasscheck__` while aiming to solve the instance.

After Fixing with Hints: Based on the reasonings, we decide to add the following hint at the end of the issue report and rerun AutoCodeRover.

> *The fix for this issue would not be located in the 'example' directory. Make sure you focus on the '__subclasscheck__' method.*

With this hint, AutoCodeRover retrieves the correct contextual information related to `__subclasscheck__` and arrives at the correct file location to be patched (Listing 4).

V. IMPLICATIONS

In our case study, we demonstrate how Watson's observed reasoning helps us identify where AutoCodeRover's implicit reasoning path went wrong. Furthermore, we show how these reasonings enable us to guide and correct AutoCodeRover's reasoning process through formulated hints. This observation is important because Agentware systems involve both Agentware developers and autonomous agents, both of which can leverage observability techniques to gain insights into the internal states and processes of Agentware. Reasoning path is crucial for Agentware developers to identify and address issues by refining meta prompts, redesigning agent interactions, and implementing other enhancements. Meanwhile, reasoning path enables the agents themselves to develop a deeper system-understanding. Allowing agents to observe and reflect on each other's reasoning paths can lead to improved quality of Agentware, as this cross-checking process enables them to refine their decision-making collaboratively. In turn, this fosters continuous improvement in Agentware's implicit reasoning capabilities and overall performance, leading to more robust and effective systems.

VI. THREATS TO VALIDITY

In this section, we discuss the threats to the validity of our study.

Internal Validity: Threats to internal validity concern factors that could have influenced our analysis and findings. One such threat in Watson arises from the fact that the surrogate agent and the primary agent are two different entities. Although both agents share the same FM architecture and decoder parameter settings, their reasoning paths may still differ. This discrepancy occurs because even with identical configurations, subtle variations in token generation or contextual interpretation can lead to different reasoning paths and consequently, divergent completions, and so the possibility of the surrogate agent producing reasoning that deviates from that of the implicit reasonings of the primary agent.

Another potential threat lies in our approach for ensuring reasoning consistency (Section III-A). Extracting recurring ideas from the reasonings generated by the surrogate agent does not guarantee the inclusion of all possible reasoning paths for the primary agent. This process may miss nuanced, less

```
Result of search_class("B"):
Found 13 classes with name B in the codebase:
They appeared in the following files:
- <file>tests/utils_tests/test_functional.py</file> (1
matches)
- <file>tests/test_runner/models.py</file> (1 matches)
- <file>tests/forms_tests/tests/tests.py</file> (1
matches)
- <file>tests/update/models.py</file> (1 matches)
- <file>tests/model_inheritance/tests.py</file> (2
matches)
- <file>tests/delete/models.py</file> (1 matches)
-
<file>tests/generic_relations_regress/models.py</file>
(1 matches)
- <file>tests/select_related_regress/models.py</file>
(1 matches)
- <file>tests/migrations/test_state.py</file> (4
matches)
```

(a) The importance of `model_inheritance/tests.py` in the primary agent's completion

```
Based on the collected context, the issue is related to the
model_inheritance/tests.py file, which contains the following
code:
class A(models.Model):
class Meta:
abstract = True
myfield = IntegerField()
class B(A):
pass
class C(A):
pass
The issue is that the fields of B and C are equal, causing a bug
when pulling the fields into a shared set. The solution is to
adjust
__eq__ so that fields with different models will compare
unequal. There also a suggestion to adjust __hash__ and
__lt__ to match
```

(b) The importance of subclasses `B` and `C` in the primary agent's completion

Fig. 4: Exemplifying the differences in attribution values among prompt tokens in `django__django-13401`


```
File "<frozen importlib._bootstrap>", line 1030, in _gcd_import
File "<frozen importlib._bootstrap>", line 1007, in _find_and_load
File "<frozen importlib._bootstrap>", line 986, in
_find_and_load_unlocked
File "<frozen importlib._bootstrap>", line 680, in _load_unlocked
File "<frozen importlib._bootstrap_external>", line 790, in exec_module
File "<frozen importlib._bootstrap>", line 228, in
_call_with_frames_removed
File "/.../example/core/models.py", line 8, in <module>
class MyModel(models.Model):
File "/.../venv/lib/python3.9/site-packages/django/db/models/base.py",
line 320, in __new__
new_class._prepare()
File "/.../venv/lib/python3.9/site-packages/django/db/models/base.py",
line 333, in _prepare
opts._prepare(cls)
File "/.../venv/lib/python3.9/site-packages/django/db/models/options.py",
line 285, in _prepare
pk_class = self._get_default_pk_class()
File "/.../venv/lib/python3.9/site-packages/django/db/models/options.py",
line 246, in _get_default_pk_class
raise ValueError(
ValueError: Primary key 'example.core.models.MyBigAutoField' referred by
DEFAULT_AUTO_FIELD must subclass AutoField.
This can be fixed in AutoFieldMeta.__subclasscheck__ by allowing
subclasses of those classes in the _subclasses property.
</issue>
```

Fig. 5: Exemplifying the differences of attribution values among prompt tokens in `django__django-14238`

frequent reasoning paths that are still valid, or it may be biased towards paths that appear more often but are not necessarily the most accurate. So, while the most common reasoning paths can emerge, their consistency cannot be fully ensured. However, this threat is partially mitigated by our approach to reasoning validation, where we compare the prompt attribution values against the reasoning to ensure their alignment with each other (Section III-A).

External Validity: Threats to external validity concern the generalization of our framework and findings. We demonstrate the usefulness of Watson in observing the implicit reasoning paths of agents by conducting a case study for AutoCodeRover to debug its reasoning paths while attempting to solve two instances of SWE-bench-lite. Future studies should extend the evaluation to various Agentware in broader context. Furthermore, while the FIM technique is ideal for our use case, our

experiment results may not be replicated for agents using FMs other than those that inherently support the FIM technique. We mitigate this threat by proposing Repetitive Chain-of-Thought Section III-A) as an alternative approach.

Construct Validity: Threats to construct validity concern the extent to which our measurements are correctly captured. In the case study conducted for AutoCodeRover, as the final step, we include a hint along with the initial instance prompt to guide the agent in finding the correct buggy locations. These hints are extracted based on the Watson-generated reasoning and the faulty decisions we identify in them, making them heavily influenced by the examiner bias. However, the purpose of the case study is to showcase the usefulness of Watson for observing implicit reasoning paths of Agentware, and not to improve the AutoCodeRover's performance. Therefore, minimizing the examiner subjectivity towards the hints is not a priority concern for the study.

VII. CONCLUSION

As Agentware become more prominent in powering complex systems, they introduce new challenges for developers, particularly in terms of observability. Unlike traditional software, where behavior can often be traced directly to explicit code assets using operational observability techniques, Agentware operate with a high degree of autonomy and unpredictability, making decisions based on evolving and dynamically retrieved contexts, learned experiences, and implicit reasoning paths that are often opaque. This opacity makes it difficult for developers to fully understand how this software function during runtime, especially when dealing with unexpected behaviors or errors, emphasizing the limitation of applying traditional operational observability techniques for Agentware.

In this work, we introduce a taxonomy of observability for Agentware, addressing the unique challenges and opportunities they present. Our taxonomy emphasizes the limitations of traditional operational observability and underscores the critical need for cognitive observability, particularly in capturing

Listing 3: The generated reasoning from Watson for Instance `django__django-14238`

```
[SURROGATE_AGENT SUMMARY] **Common Points Across Reasonings:**
1. **Focus on "MyModel" and '__init__' Method**: Multiple explanations consistently mention the class "
   MyModel" and its constructor method '__init__', indicating that they are central to identifying the
   bug locations.
2. **DEFAULT_AUTO_FIELD Issues**: There is a recurring theme regarding problems with '
   DEFAULT_AUTO_FIELD', suggesting it is linked to the bugs being analyzed.
3. **File Location**: The file "example/core/models.py" is frequently identified as a potential
   source of issues, reinforcing its importance in diagnosing the problem.
4. **Need for Additional Context**: Several reasonings highlight whether more context or information
   from other files/methods might be necessary, reflecting uncertainty about fully understanding all
   aspects of the issue.

**Unique Perspectives:**
- Some explanations assert that no further context is needed (Explanations 8), while others suggest
  making additional search API calls (Explanations 6 and 7) to gather more information before
  finalizing conclusions about bug locations.

In summary, there's strong consensus around focusing on "MyModel", particularly its initialization
  method and how it interacts with 'DEFAULT_AUTO_FIELD' within "example/core/models.py". However,
  opinions differ on whether additional context gathering through searches would be beneficial or if
  sufficient information has already been acquired to proceed with solutions.
```

Listing 4: A segment of the output from AutoCodeRover reapplied on `django__django-14238` with the reasoning hint

```
[ASSISTANT] Based on the search results, we can see that there are 12 classes with the name MyModel in
   the codebase, but none of them are located in the "example" directory. This means that the issue
   is not related to the "example" project.
We also found 1 method with the name __subclasscheck__, which is located in the django/db/models/fields
   /__init__.py file. This method is inherited from the AutoFieldMeta class and is used to check if a
   given subclass is a subclass of AutoField.
Additionally, we found 3 snippets containing 'DEFAULT_AUTO_FIELD =' in the codebase. These snippets are
   located in the tests/test_sqlite.py, tests/check_framework/test_model_checks.py, and django/conf/
   global_settings.py files. These snippets are used to set the default primary key field type in the
   codebase.
Finally, we found 2 snippets containing '__subclasscheck__' in the codebase, both of which are located
   in the django/db/models/fields/__init__.py file. These snippets are used to check if a given
   subclass is a subclass of AutoField.
Based on this context, we can conclude that the issue is related to the django/db/models/fields/
   __init__.py file, specifically the AutoFieldMeta class and its __subclasscheck__ method. The fix
    for this issue would not be located in the "example" project, but rather in the django project
   itself.
```

agent implicit reasoning paths. To begin to address these challenges, we propose a novel framework called Watson, which focuses on observing an agent's implicit reasoning, providing deeper insights for more effective monitoring and debugging of Agentware. We validate the efficacy of our approach using the state-of-the-art Agentware, AutoCodeRover, demonstrating how cognitive observability enhances agent performance and significantly improves debuggability, while also highlighting the ability for Agentware to leverage these observability insights for multi-agent debates and reflections and in turn correct potential errors in runtime. Together, these contributions pave the way for more resilient, transparent, and better performing Agentware.

REFERENCES

[1] *A complete guide to LLM observability with Open-Telemetry and Grafana Cloud.* last accessed: 2024-10-10. URL: https : / / grafana . com / blog / 2024 / 07 / 18 / a - complete - guide - to - llm - observability - with - opentelemetry-and-grafana-cloud/.

[2] OpenAI et al. *GPT-4 Technical Report.* 2024. URL: https://arxiv.org/abs/2303.08774.

[3] *AutoCodeRover Code Repository.* last accessed: 2024-10-08. URL: https://github.com/nus-apr/auto-code-rover.

[4] Mohammad Bavarian et al. *Efficient Training of Language Models to Fill in the Middle.* 2022. URL: https://arxiv.org/abs/2207.14255.

[5] Maciej Besta et al. "Graph of Thoughts: Solving Elaborate Problems with Large Language Models". In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence.* 2024.

[6] Yupeng Chang et al. "A Survey on Evaluation of Large Language Models". In: *ACM Transactions on Intelligent Systems and Technology* 15.3 (2024).

[7] *ChatGPT*. last accessed: 2024-10-09. URL: https://chatgpt.com/.

[8] Boyuan Chen et al. "Towards training reproducible deep learning models". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022.

[9] *Concepts - Tracing*. last accessed: 2024-10-10. URL: https://docs.smith.langchain.com/concepts/tracing.

[10] *Copilot Internals*. last accessed: 2024-10-09. URL: https://thakkarparth007.github.io/copilot-explorer/posts/copilot-internals.

[11] Sunhao Dai et al. "Bias and Unfairness in Information Retrieval Systems: New Challenges in the LLM Era". In: *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2024.

[12] P. A. M. Dirac. *The Principles of Quantum Mechanics*. 4th. Oxford: Clarendon Press, 1958.

[13] *django Web Framework*. last accessed: 2024-10-08. URL: https://www.djangoproject.com/.

[14] Ximing Dong et al. *PromptExp: Multi-granularity Prompt Explanation of Large Language Models*. under review, preprint available on arXiv. 2024.

[15] *Dynatrace*. last accessed: 2024-10-08. URL: https://www.dynatrace.com/.

[16] Daniel Fried et al. "InCoder: A Generative Model for Code Infilling and Synthesis". In: *Proceedings of the 11th International Conference on Learning Representations*. 2023.

[17] *GitHub Copilot*. last accessed: 2024-10-09. URL: https://github.com/features/copilot.

[18] Linyuan Gong et al. "Evaluation of LLMs on Syntax-Aware Code Fill-in-the-Middle Tasks". In: *Proceedings of the 41st International Conference on Machine Learning*. 2024.

[19] Siyuan Guo et al. "DS-Agent: Automated Data Science by Empowering Large Language Models with Case-Based Reasoning". In: *Proceedings of the 41st International Conference on Machine Learning*. 2024.

[20] *Haptik, Drive Business Efficiency at Scale with Generative AI*. last accessed: 2024-10-08. URL: https://www.haptik.ai/.

[21] Ahmed E. Hassan, Zhen Ming Jiang, and Yasutaka Kamei. *FM+SE Summit 2024*. Accessed: 2024-10-09. 2024. URL: https://fmse.io/index.html.

[22] Ahmed E. Hassan et al. *FM+SE Vision 2030*. Accessed: 2024-10-09. 2023. URL: https://fmse.io/vision/index.html.

[23] Ahmed E. Hassan et al. "Rethinking Software Engineering in the Era of Foundation Models: A Curated Catalogue of Challenges in the Development of Trustworthy FMware". In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 2024.

[24] Ahmed E. Hassan et al. *Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap*. 2024. URL: https://arxiv.org/abs/2410.06107.

[25] *Helicone*. last accessed: 2024-10-08. URL: https://www.helicone.ai/.

[26] Shengxin Hong et al. *ArgMed-Agents: Explainable Clinical Decision Reasoning with LLM Disscusion via Argumentation Schemes*. 2024. URL: https://arxiv.org/abs/2403.06294.

[27] Sirui Hong et al. "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework". In: *Proceedings of the 12th International Conference on Learning Representations*. 2024.

[28] Shengran Hu, Cong Lu, and Jeff Clune. *Automated Design of Agentic Systems*. 2024. URL: https://arxiv.org/abs/2408.08435.

[29] Lei Huang et al. *A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions*. 2023. URL: https://arxiv.org/abs/2311.05232.

[30] *Humanloop*. last accessed: 2024-10-08. URL: https://humanloop.com/.

[31] *Introducing Devin, the first AI software engineer*. last accessed: 2024-10-08. URL: https://www.cognition.ai/blog/introducing-devin.

[32] *Introducing OpenAI o1*. last accessed: 2024-10-08. URL: https://openai.com/o1/.

[33] M. Izadi et al. "Language Models for Code Completion: A Practical Evaluation". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024.

[34] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. URL: https://arxiv.org/abs/2310.06770.

[35] Foutse Khomh et al. *Software Engineering for Machine Learning Applications (SEMLA)*. Accessed: 2024-10-09. 2023. URL: https://semla-polymtl.github.io/semla2023.

[36] *Langfuse*. last accessed: 2024-10-08. URL: https://langfuse.com/.

[37] *LangSmith by LangChain*. last accessed: 2024-10-08. URL: https://www.langchain.com/langsmith/.

[38] *Learning to Reason with LLMs*. last accessed: 2024-10-08. URL: https://openai.com/index/learning-to-reason-with-llms/.

[39] Patrick Lewis et al. "Retrieval-augmented generation for knowledge-intensive NLP tasks". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. 2020.

[40] B. Li et al. "Enjoy your observability: an industrial survey of microservice tracing and analysis". In: *Empirical Software Engineering* 27.1 (2022).

[41] Hao Li, Cor-Paul Bezemer, and Ahmed E Hassan. *Software Engineering and Foundation Models: Insights from Industry Blogs Using a Jury of Foundation Models*. under review, preprint available on arXiv. 2024.

[42] Xiao-Hui Li et al. "A Survey of Data-Driven and Knowledge-Aware eXplainable AI". In: *IEEE Transactions on Knowledge and Data Engineering* 34.1 (2022).

[43] Xiaonan Li et al. "LLatrieval: LLM-Verified Retrieval for Verifiable Generation". In: *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 2024.

[44] Jiaying Lu et al. "AgentLens: Visual Analysis for Agent Behaviors in LLM-based Autonomous Systems". In: *IEEE Transactions on Visualization and Computer Graphics* (2024).

[45] *Lunary*. last accessed: 2024-10-08. URL: https://lunary.ai/.

[46] Tula Masterman et al. *The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey*. 2024. URL: https://arxiv.org/abs/2404.11584.

[47] *Nebuly - Explicit and Implicit LLM User Feedback Quickguide*. last accessed: 2024-10-08. URL: https://www.nebuly.com/blog/explicit-implicit-llm-user-feedback-quickguide.

[48] *Open Platform for Enterprise AI*. last accessed: 2024-10-09. URL: https://opea.dev/.

[49] *OpenTelemetry*. last accessed: 2024-10-08. URL: https://opentelemetry.io/.

[50] Guglielmo Papagni et al. "Artificial agents' explainability to support trust: considerations on timing and context". In: *AI & Society* 38.2 (2022).

[51] Joon Sung Park et al. "Generative agents: Interactive simulacra of human behavior". In: *Proceedings of the 36th ACM Symposium on User Interface Software and Technology*. 2023.

[52] Mauro Pezzè. *2030 Software Engineering*. Accessed: 2024-10-09. URL: https://conf.researchr.org/home/2030-se.

[53] Mauro Pezzè et al. *The Trailer of the ACM 2030 Roadmap for Software Engineering*. Accessed: 2024-10-09. 2024. URL: https://www.inf.usi.ch/faculty/pezze/media/SE2030SENreport.pdf.

[54] *Phoenix by Arize*. last accessed: 2024-10-08. URL: https://phoenix.arize.com/.

[55] *Qwak - LLMops*. last accessed: 2024-10-08. URL: https://www.qwak.com/products/llmops.

[56] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. URL: https://arxiv.org/abs/2303.11366.

[57] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: http://research.google.com/archive/papers/dapper-2010-1.pdf.

[58] *SWE-bench Lite: A Canonical Subset for Efficient Evaluation of Language Models as Software Engineers*. last accessed: 2024-10-08. URL: https://www.swebench.com/lite.html.

[59] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. URL: https://arxiv.org/abs/2302.13971.

[60] *Traceloop*. last accessed: 2024-10-08. URL: https://www.traceloop.com/.

[61] *Traceloop - OpenLLMetry*. last accessed: 2024-10-08. URL: https://www.traceloop.com/openllmetry.

[62] Ashish Vaswani et al. "Attention is all you need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017.

[63] Avish Vijayaraghavan and Cosmin Badea. "Minimum levels of interpretability for artificial moral agents". In: *AI and Ethics* (2024).

[64] Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. 2022.

[65] *Weights and Biases - Weave*. last accessed: 2024-10-08. URL: https://wandb.ai/site/weave/.

[66] *WhyLabs*. last accessed: 2024-10-08. URL: https://whylabs.ai/.

[67] Qingyun Wu et al. *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation*. 2023. URL: https://arxiv.org/abs/2308.08155.

[68] Fangzhi Xu et al. *Are Large Language Models Really Good Logical Reasoners? A Comprehensive Evaluation and Beyond*. 2024. URL: https://arxiv.org/abs/2306.09841.

[69] Shunyu Yao et al. "Tree of thoughts: deliberate problem solving with large language models". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. 2024.

[70] Eric Zelikman et al. "STaR: Bootstrapping reasoning with reasoning". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. 2022.

[71] Xijia Zhang et al. *Explaining Agent Behavior with Large Language Models*. 2023. URL: https://arxiv.org/abs/2309.10346.

[72] Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. *On the Diagram of Thought*. 2024. URL: https://arxiv.org/abs/2409.10038.

[73] Yifan Zhang et al. *Cumulative Reasoning with Large Language Models*. 2024. URL: https://arxiv.org/abs/2308.04371.

[74] Yuntong Zhang et al. "AutoCodeRover: Autonomous Program Improvement". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024.

[75] Zhixue Zhao and Boxuan Shan. *ReAgent: A Model-agnostic Feature Attribution Method for Generative Language Models*. 2024. URL: https://arxiv.org/abs/2402.00794.

[76] Li Zhenzhu et al. "GPT-agents based on medical guidelines can improve the responsiveness and explainability of outcomes for traumatic brain injury rehabilitation". In: *Scientific Reports* 14.1 (2024).