Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Hypothesis-Driven Debugging with Large Language Models

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Daniel Passson
geboren am: 30.12.1998
geboren in: Heppenheim

Gutachter/innen: Prof. Dr. Lars Grunske
Dr. Thomas Vogel

eingereicht am: ..... verteidigt am: .....

Abstract. Why does a program fail for one input but work as expected for another? Eberlein et al. proposed *Avicenna* [10], a tool aiming to answer this question by using machine learning to automatically identify the failure-inducing input features of buggy programs. Inspired by their approach, we propose *HDDAgent*, a **h**ypothesis-**d**riven **d**ebugging tool using **agent**ic workflows to leverage the problem-solving and reasoning capabilities of large language models (LLMs). Using a context-free grammar as input specification and a set of initial inputs, *HDDAgent* invents a hypothesis as to the failure-inducing input features and uses grammar-based fuzzing with constraints derived from the hypothesis to generate additional program inputs. Testing the program with this new set of inputs allows the LLM-based learner to evaluate the validity of the initial hypothesis and to iteratively refine it, resulting in a complete and concise diagnosis of the exact failure conditions of the program under test. Our evaluations showed that *HDDAgent* was able to produce exact diagnoses for 9 out of 10 benchmark subjects, outperforming *Avicenna*.

Contents

1 Introduction

Software bugs can be elusive. [35] At times a program may work just as expected but fail at other times, making the issue difficult to debug. Knowing when a program fails can be vital when debugging software, but identifying the exact failure conditions is often easier said than done and, more importantly, time-consuming. While automatic testing methods such as random fuzzing are easy to deploy, they also lack the capacity to properly test programs with complex input specifications as they often generate invalid inputs. [34]

To address this problem, Kampmann et al. proposed *Alhazen* [16] as a method to automatically identify which features of the program input are related to the failure. To that end *Alhazen* employs a decision tree learner that identifies the input features that are most relevant to the program failure. The learned associations form hypotheses which are then refined or refuted based on additional inputs that *Alhazen* generated based on the program's grammar. By repeating this process, their tool is able to find a theory explaining the failure that can then be used as a predictor for failures if given an input or as a producer of failing inputs that can be used to aid in debugging. This idea was later refined by Eberlein et al. when they proposed *Avicenna* [10]. Using a machine learning based learner, *Avicenna* is capable of identifying semantic relationships between input elements whereas *Alhazen* can only associate individual features of the input in isolation.

With this thesis we aim to further enhance this hypothesis-driven approach, using large language models to invent, test and refine hypotheses. We will present an agentic workflow that leverages the strong reasoning and tool-calling capabilities of modern LLMs and compare the effectiveness of our method to *Avicenna* using a benchmark suite of 10 subjects. Finally, we will perform a qualitative analysis of the reasoning output of the LLM to evaluate their proficiency at scientific debugging tasks.

RQ1 How effective is *HDDAgent* at identifying the exact failure conditions of programs compared to *Avicenna*?

RQ2 How effectively do LLMs reason about failure conditions and to what extent does it align with scientific debugging practices?

We summarize the contributions of this thesis as follows:

- We propose *HDDAgent*, an LLM-based tool for diagnosing a programs failure conditions.
- We evaluate the effectiveness of our approach at producing exact diagnoses, comparing it to *Avicenna*.
- To that end we present a way of categorizing diagnoses based on their location in the program's input space relative to the ground truth failure conditions.

- We perform a qualitative analysis of the reasoning content produced by an LLM running our agentic workflow.

This thesis is structured as follows. First we present the relevant theoretical concepts in section 2. Then we introduce the related work in section 3. In section 4 we present our approach followed by a simple example. We describe our methodology in section 5 followed by the results from our evaluation in section 6 5. In section 7 we discuss our findings. Finally, we conclude this thesis in section 8, summarizing our results and contributions.

2 Background

We propose a hypothesis-driven approach that uses Large Language Models (LLMs) to diagnose program failures by identifying the relevant input features that lead to such failures. We implement an agentic optimizer-evaluator workflow to automatically invent and refine relevant hypotheses using a grammar-based fuzzer to generate black-box tests. In this section we will introduce key concepts that our approach builds upon.

2.1 Scientific Debugging

Borrowing from the *scientific method*, the basis of all experimental science which describes the process of obtaining a *theory* that explains some aspect of the universe, *scientific debugging* is a technique for systematically finding a **diagnosis** explaining the failure of a program. [35]

Step	**Scientific Method**	**Scientific Debugging**
1.	Observe some aspect of the universe.	Observe the failure of a program.
2.	Invent a hypothesis consistent with the observation.	Invent a hypothesis consistent with the observation.
3.	Use the hypothesis to make predictions.	Use the hypothesis to make predictions.
4.	Test the predictions using experiments and further observations and modify the hypothesis according to the results.	Test the predictions using experiments: • If the experiment satisfies the predictions, refine the hypothesis. • Otherwise, create an alternative hypothesis.
5.	Repeat steps 3 and 4 until there are no discrepancies between the hypothesis and the observation.	Repeat steps 3 and 4 until the hypothesis can no longer be refined.

Table 1: Comparison of the scientific method and scientific debugging.

The process of scientific debugging is similar to that of the scientific debugging method as can be seen in Table 1 with steps 2 and 3 being identical and step 1 differing only in the scope of our observations. Steps 4 and 5 differ in that we still choose to refine our hypothesis if the experiment satisfies our predictions when using scientific debugging while the scientific method finishes once there are no more discrepancies between the hypothesis and the observation. That is because of the narrower scope of hypotheses when using scientific debugging. As we debug, we tend to focus on small aspects of

the program at a time. A confirmed hypothesis may therefore only partially relate to the observed failure and multiple iterations resulting in many valid hypotheses might be necessary. It is therefore important that any refined hypothesis must include all earlier hypotheses that have been confirmed as well as exclude all hypotheses that have been rejected. This ensures that our refined hypotheses are consistent with all prior observations.

2.2 Grammar-Based Fuzzing

While it is possible to write programs that only perform one specific task, oftentimes some degree of additional flexibility is desired. Many programs are therefore designed to accept a set of inputs to be processed by the program. But as the flexibility increases, so does the effort required for proper testing, because the execution path may now depend on the chosen test inputs. For this reason, **input generators**, programs that output the inputs for other programs, are often used in automated software testing. [34] Generators that randomly generate inputs for the program under test are called **fuzzers**. While simple random fuzzing has been shown to be effective at finding severe bugs [26], most inputs produced in this fashion will be invalid, thus merely the program's ability to sanitize its inputs is being tested. [34] In order to effectively test complex programs, one therefore needs a way to only generate inputs that are valid according to the program's *input specification*. One such way is the use of **grammar-based Fuzzers**, that use (typically context-free) grammars as input specification. Additionally, newer approaches allow for *constraints over non-terminals* to further shape the generated inputs.

2.3 Hypothesis-Driven Debugging

Understanding when programs fail is often an important step in understanding why they fail and finally how to fix them. *Avicenna* [10] is a technique aiming to automatically determine the failure conditions of programs. It builds on *Alhazen* [16], an earlier approach that uses a decision tree learner to learn which input features are associated with the program's failure and generate new test inputs in order to iteratively improve the failure model. Both methods are hypothesis-based (see subsection 2.1) and examine the program under test in a **black-box setting**, that is just by observing the programs output for each generated input without relying on additional runtime information or the programs source code. They both require a context-free grammar and a set of initial inputs (at least one of them failing) in order to operate. By leveraging *ISLearn* [31], a pattern-based approach for learning constraints, *Avicenna* extends the capabilities of *Alhazen* to also learn semantic properties of the inputs. It is therefore capable of diagnosing complex failures like the infamous *Heartbleed* bug, a vulnerability in the TLS Heartbeat protocol that is triggered when the *length* parameter would exceed the actual length of the given *payload*. *Alhazen* would only identify failure circumstances relating to the input features in isolation, such as:

$$\text{len}(\langle payload \rangle) \leq 16357 \tag{1}$$

Avicenna, on the other hand correctly identifies the relation between $\langle length \rangle$ and $\langle payload \rangle$:

$$\text{int}(\langle length \rangle) > \text{len}(\langle payload \rangle) \tag{2}$$

2.4 Large Language Models

Large language models (LLMs) are neural network based language models that are trained on vast amounts of text data and feature model sizes often well in excess of a billion parameters. [27, 36] They are usually trained to simply predict the next token during the pre-training stage and then later fine-tuned to improve their instruction-following capabilities as well as align them using human feedback in order to provide more accurate, helpful and harmless responses. In recent years, LLMs have become a major area of research in machine learning with applications ranging from natural language processing to AI assistants. **Chain-of-thought (CoT)** refers to a variety of techniques aiming to improve the problem-solving capabilities of LLMs through human-like reasoning. Early approaches revolved around specialized prompts forcing the LLM to solve its task step-by-step while providing explanations of its reasoning. Recently however, fine-tuning and reinforcement learning are being used to create LLMs with strong reasoning abilities without the need of special prompting. [32].

2.5 Agentic Systems

While LLMs by themselves are already capable of solving many problems on their own, depending on the task the use of traditional approaches is preferable or even necessary. **Tool-calling** is an approach to combine the best of both worlds. Leveraging their ability to output structured responses, the LLMs are given access to a set of tools to aid them in their task. The LLM may autonomously select tools and provide arguments when applicable and use the results to generate more accurate responses. This capability has given rise to a new paradigm: **Agentic Systems**. [30]

With **agentic workflows**, the execution follows predefined code paths. Prompt-chaining involves using multiple LLM calls in succession, often with additional pre- or post-processing steps in between. With Parallelization, multiple LLMs are queried concurrently. The response is then processed by an aggregator LLM.

Some workflows allow an LLM to choose between code branches. For the routing pattern an LLM, called a router in this context, is given a problem statement and will then select the path it deems appropriate. It resembles the orchestrator pattern with the key difference being that the orchestrator may select multiple paths at once. Evaluator-optimizer workflows involve a generator node that proposes a solution and an evaluator may approve or reject the solution and provide additional feedback. This workflow allows the LLM to iteratively improve its response.

Agents provide the highest degree of autonomy as the LLM is given full control over the problem-solving process.

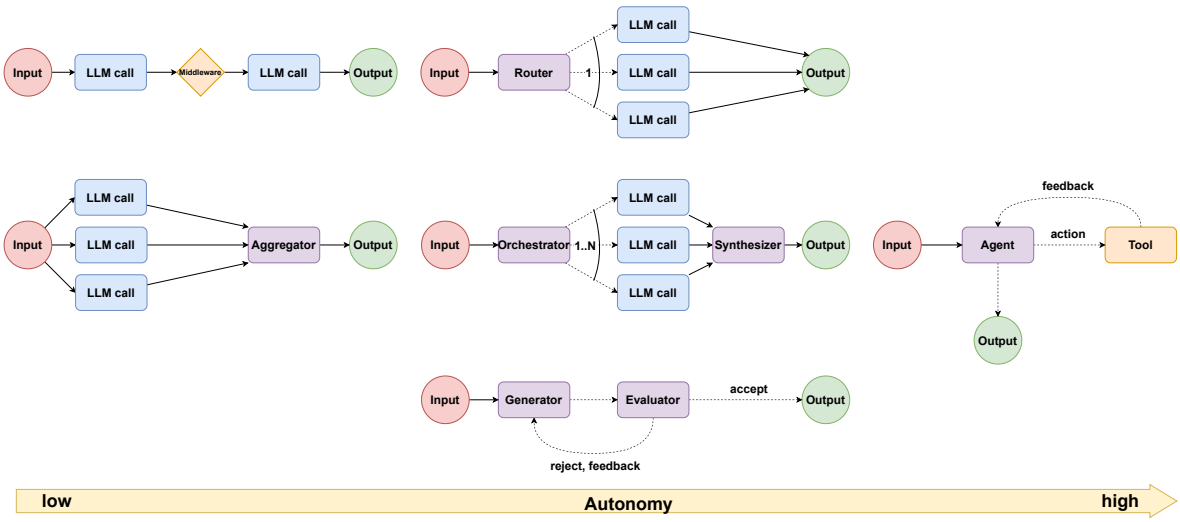

Figure 1: Types of agentic systems and their relative autonomy

It is important to note that the presented design patterns may be modified or even combined. For example, an orchestrator could be used in combination with several agents specialized for specific tasks.

3 Motivation and Related Work

With this thesis we intend to build a hypothesis-driven debugging tool that automatically identifies the relevant input features that lead to the failure of buggy programs. Our setting is that of *Alhazen* and *Avicenna*, where we iteratively invent and refine hypotheses based on black-box tests returning passing or failing as a result. Inspired by *Avicenna*'s workflow we plan to leverage the knowledge and reasoning capabilities of state-of-the-art LLMs as a replacement for *Avicenna*'s classical ML-based learner using the *Fandango* grammar-based fuzzer to generate the inputs necessary for refining the hypotheses. In this section, we will explore the related work and discuss similarities as well as differences from our work. Categorizing works based on their goal and the type of methods used, we visualize the results of this discussion in Figure 2.

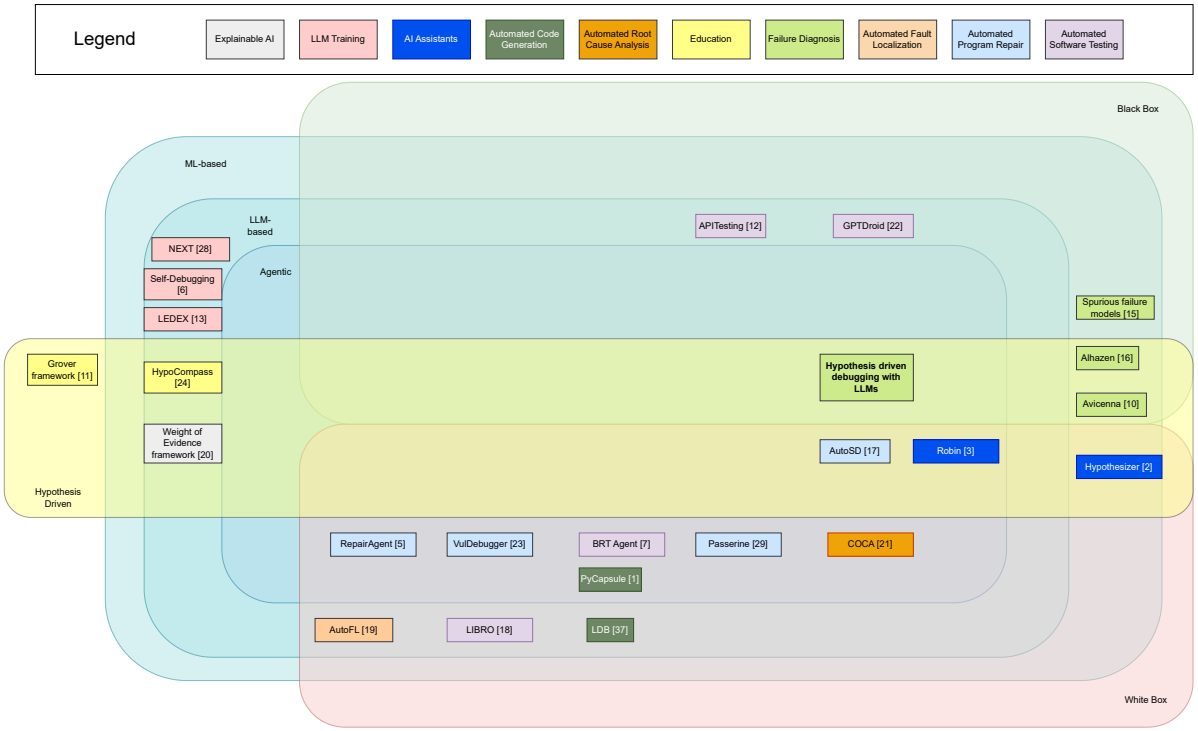

Figure 2: Our approach in comparison to related work

In 2017, Grover et al. [11] presented a theoretical framework that uses a **hypothesis-driven approach** to complement existing data-driven analytics in order to assess students' computational thinking skills. Ma et al. [24] present another application of hypothesis-driven approaches in an educational setting. They propose *HypoCompass*, an LLM-based tutoring agent that helps students improve their debugging skills by interactively guiding them to solve debugging-related tasks. Through simulated conversations with LLM-agents, students are tasked to invent hypotheses on program failures, write test cases reflecting their hypotheses and finally explain the program failure in natural language. This process closely resembles the **scientific debugging principles** that our approach is based on. However instead of using LLM-agents to teach students, we use LLMs to

explain program failures automatically in an automated software engineering setting. Another hypothesis-driven approach using LLMs is proposed by Le et al. [20]. Their *Weight of Evidence* framework aims to assist users with decision-making tasks while withholding explicit recommendations. They argue that AI recommendations have been shown to hinder human decision-making in some cases and therefore focus on providing evidence-informed hypotheses to the user instead. While they don't use their framework for software engineering tasks, the use of hypothesis-driven approaches in explainable AI is highly relevant to debugging tasks.

Large Language Models have undergone a series of advancements over the past years. In particular, improving their ability to self-debug has been a focal point of researchers with earlier solutions relying on few-shot prompting [6] while newer approaches focus on improving their chain-of-thought reasoning for debugging tasks by fine-tuning the model. [13, 28]

Automatic Fault Localization is one of those areas that have benefited from the rise of LLMs. [25] Proposed by Kang et al. [19], *AutoFL* is a tool that leverages tool calling to allow an LLM to autonomously retrieve information from a program's source code in order to find the bug location as well as an explanation of the root cause. Unlike our approach, *AutoFL* is a static white-box method, requiring access to the programs source code. Furthermore the explanations it provides relate to errors in the code while we aim to diagnose failure conditions by identifying relevant properties of the program's inputs. With advancements in LLMs' tool-calling capabilities, autonomous agents have been employed for a variety of tasks. *RepairAgent* [5] is one such example, where an LLM is guided to autonomously use tool calls for automated program repair. *VulDebugger* [23] is an LLM-based debugging agent designed to automatically localize and repair program vulnerabilities. By executing a Proof of Concept triggering the vulnerability, *VulDebugger* is able to extract information about the vulnerability, including its location, using vulnerability-free constraints to trigger a crash when the vulnerability occurs. Based on the crash message and the source code, the LLM is then directed to place breakpoints allowing it to collect dynamic information about the program state. By then comparing the actual state to the expected state inferred from the vulnerability-free constraints, *VulDebugger* can then localize the vulnerability, identify its root cause, generate a patch and validate the fix by testing whether the vulnerability is still triggered or not. While both, *VulDebugger* and our approach are LLM-based and make use of tool-calling, their agent has a greater degree of autonomy compared to our approach which follows a stricter agentic workflow. Furthermore, their goal is **automated vulnerability repair** in a white-box scenario while we attempt automatic bug diagnosis in a black-box setting.

Kang et al. [18] use LLMs to automatically generate bug reproduction tests. Their tool, *LIBRO*, prompts an LLM with varying prompts to produce a number of candidate tests. After postprocessing the LLM's response the candidates are then evaluated in order to rank and select the candidates that are to be suggested to the user. At Google, Cheng et al. [7] propose *BRT Agent*, an LLM-agent that automatically generates bug reproduction tests from user-submitted bug reports. The agent autonomously inspects the program's source code to locate possible bug locations, writes tests to trigger said bug and execute them while observing the result. Rondon et al. [29], another team of

Google researchers, discuss the viability of agent-based program repair approaches in an enterprise setting and present *Passerine*, their own repair agent that autonomously repairs bugs from human- or machine-generated bug reports using only a small set of provided tools. Both agents by Google differ in their use of LLMs from our approach in that they choose a highly autonomous ReAct [33] style agent design over a more restrictive agentic workflow and their goals, automatic bug reproduction and patch generation respectively, differ from ours as well. Automated root cause analysis is another application of LLMs with *COCA* [21] being one such example. It extracts relevant code snippets based on issue reports and reconstructs relevant execution paths. Using both, code knowledge and stack traces collected at runtime, *COCA* localizes the bug and provides a summary of its root cause in natural language, similar to how our approach produces a natural language diagnosis. Both approaches also use a more restrictive workflow rather than a fully autonomous agent, although *COCA* uses a four-phase design using prompt chaining, while our approach employs an evaluator-optimizer design. *COCA* however uses code knowledge and stack traces and the summaries it produces are aimed to explain the internal reasons of the program's failure while we propose a black-box approach, merely requiring information on whether our generated tests are passing or failing and the diagnoses we produce are focused on the properties of the inputs leading to the program's failure and not its root cause. Lastly, our approach is hypothesis-driven, using scientific programming principles to prompt the LLM to invent and test hypotheses while *COCA* guides the LLM's reasoning without any use of hypotheses.

Kang et al. [17] propose *AutoSD*, an approach that uses automatic scientific debugging to generate explainable patches. Like our approach, *AutoSD* consists of an evaluator-optimizer workflow to invent and test hypotheses. Both approaches differ however in their goal and setting. While *AutoSD* is a white-box technique for automated program repair, we explain program failures in a black-box setting. Another difference is that *AutoSD* concludes the debugging process once it finds a hypothesis that is supported by its observations, while in our case the LLM may choose to continue the optimization loop in order to find a more concise hypothesis.

LDB [37] is a debugging framework developed by Zhong et al. that allows LLM-based code generators to iteratively generate and validate code block by block rather than generate the entire program in a single pass. Their approach is iterative, akin to our optimizer-evaluator workflow, however *LDB* does not utilize tool calling but rather repeatedly queries the LLM, providing additional information acquired by the framework. Another technique for automatic code generation is proposed by Adnan et al. [1]. Their framework, *PyCapsule*, employs a two-agent pipeline consisting of a programmer agent that generates the code and handles debugging and an executor agent that validates the generated code and extracts error information in case of a failure. This architecture resembles the evaluator-optimizer pattern our approach uses, although with complex agents instead of the simpler nodes we use.

Alaboudi and Latoza [2] propose *Hypothesizer*, an interactive debugging tool aiming to assist developers in finding relevant hypotheses while debugging GUI-based web applications. After the developer demonstrated the issue in an instrumented browser session, *Hypothesizer* selects relevant hypotheses from a set of predefined hypotheses by

comparing their conditions with the recorded program behavior using pattern-matching. The developer is then asked to describe the issue by selecting predefined labels to further narrow down the selection of hypotheses. *Hypothesizer* then provides the developer with an investigation plan to collect further evidence in order to test the remaining hypotheses. Finally, once a hypothesis has been confirmed, the developer is provided with step-by-step instructions on how to fix the defect while also suggesting a location for the fix. A team at Microsoft, consisting of Bajpai et al. [3] discussed the tendency of current AI-agents to leap into action when given insufficient context and propose *Robin*, a conversational agent built on GitHub Copilot. They introduced the investigate & respond pattern which focuses on gathering required information automatically from the IDE or interactively from the developer before responding. To achieve this, they employ four agents in an orchestrator workflow.

Other applications like GUI testing [22] leverage the reasoning and decision-making capabilities of LLMs. Huynh et al. [12] propose *APITesting*, a framework that uses an LLM to mine and test API constraints in a black-box setting. From the natural language description provided by the API specification the LLM extracts the constraints and generates test cases to identify matches and mismatches between the mined constraints and the API's behavior. Jodat et al. [15] propose an ML-based approach to build failure models to identify spurious failures, that is failures induced by invalid or unrealistic test inputs. By automatically generating test inputs they train a surrogate model in order to identify the regions of the input space that lead to spurious failures. Like our approach, they aim to explain failures with respect to the input features that trigger them by generating and running test inputs in a black-box setting. However, while their approach contains an optimization loop in order to train the surrogate model, this loop is not hypothesis-driven. Furthermore they aim to identify spurious failures in particular and focus on systems with numeric inputs while our approach focuses on systems with string-based inputs defined by an input grammar without restricting the type of failures we want to explain.

4 Approach

Following good practices for designing LLM-based workflows [30], we developed an *agentic workflow* resembling the *evaluator-optimizer pattern*. We opted for this design for two main reasons.

1. *Avicenna*'s design already resembles a loop where hypotheses are iteratively refined until refinement is no longer possible or reaching the maximum amount of iterations. Maintaining this design similarity allow us to better understand the impact of using LLMs instead of traditional machine learning methods.

2. A simpler prompt-chaining approach wouldn't allow the LLM to play an active role in the control flow while a more autonomous agent design bears the risk of getting lost in sub-tasks. As we have a well-defined task that only requires LLM-based decision-making at a few points of the workflow, settling for the middle ground seems like the best option.

Our workflow as illustrated in Figure 3 starts by evaluating the initial inputs against the oracle and passing the result to the *learner* node, which is tasked with inventing a hypothesis as to the failure conditions of the program under test. The hypothesis is then used to generate a list of constraints that failing inputs as predicted by the hypothesis should satisfy. This process is divided into two parts, first generating the constraints in natural language and then formalizing them. Then, the *generator* node is able to call the *Fandango fuzzer* [34] , which returns a list of new program inputs. In addition to inputs that satisfy the previously generated constraints, the list will also include inputs generated using negated constraints to serve as counterexamples to the hypothesis. The *generator* may choose to call the Fandango fuzzer as many times as it deems necessary. After submitting the new set of test inputs, the *generator* passes them to the oracle which runs the program under test against each input and evaluates the result as either *PASSING*, *FAILING*, or *UNDEFINED*. The results are then returned to the learner which uses them to refine the hypothesis. In the end, the learner decides whether to continue iterating or to return the final diagnosis if it cannot be further refined.

We will now explain the design of each node in more detail.

4.1 Learner

The *learner* is the starting node of our workflow and at the same time the node that will return the final diagnosis. In its system prompt we include a high-level explanation of scientific debugging as explained in section 2.1 as well as a few examples of valid and invalid hypotheses. We then give a brief explanation of the *learner*'s role in the workflow while also mentioning the tasks that will be performed by other nodes followed by a step-by-step workflow in hopes of guiding the LLM to perform explicit and verbose reasoning with regard to its observations and deductions. Using structured output, the learner can either submit a hypothesis, continuing the optimization loop, or return the final diagnosis, thus terminating the workflow. The LLM is instructed to only return the diagnosis if it is certain that the hypothesis cannot be further refined.

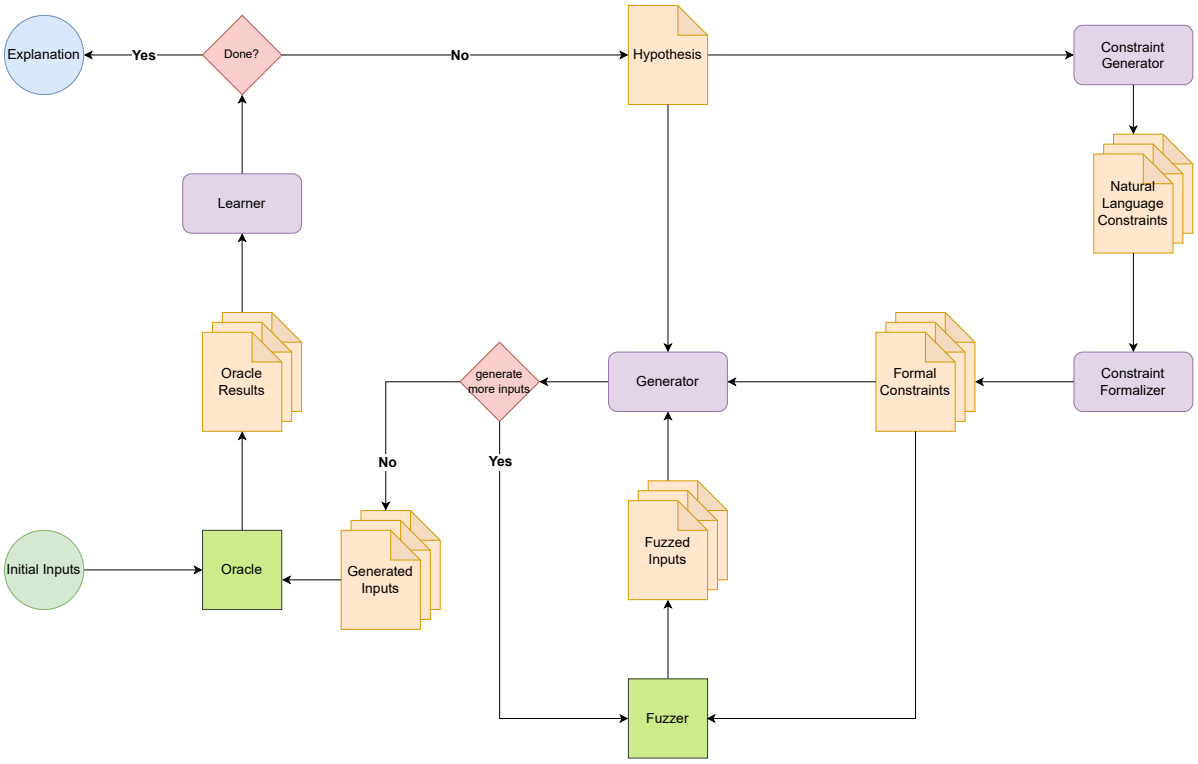

Figure 3: Overview of our proposed workflow. LLM-nodes are marked in purple, control flow branches are marked in red, green squares indicate middleware and data is marked in orange.

4.2 Constraint Generation and Formalization

The *constraint generator* receives a hypothesis from the *learner* and submits a structured list of natural language constraints. In its prompt we included a short description of scientific debugging as well as a task description including a few examples. Lastly, we provide step-by-step instructions to first analyze the grammar and identify the symbols relevant to the hypothesis, consider how the grammar needs to be restricted in order to generate inputs relevant to the hypothesis and then write down these constraints, ensuring they are concise and satisfiable.

For the *constraint formalizer*, we use a similar prompt, altering the problem statement to take the previously generated natural language constraints as input to then output a list of formal constraints. We also include an excerpt of relevant sections of the *Fandango* documentation [34] as well as additional instructions on how to escape certain control characters. The formal constraints are then submitted as a structured list of strings.

From our preliminary testing we identified the constraint generation as a critical phase prone to various types of errors. Before moving on with our workflow, we therefore validate the constraints by calling *Fandango*'s `fuzz` method once while catching any exceptions the fuzzer may throw. In the event of any error, we retry the constraint generation and formalization, hoping for the nondeterministic nature of LLMs to result

in slightly different constraints that pass the validation. We allow for up to 3 retries before moving on with the workflow regardless.

4.3 Generator

The *generator* is given the hypothesis and the previously generated formal constraints in order to produce a list of new inputs for the program under test. In the system prompt we include a brief task definition including a description of the expected input and output of the node as well as an explanation of the *Fandango fuzzer*, which the LLM can call as a tool. The step-by-step instructions at the end of the system prompt encourage the LLM to reference the grammar to reason about the input features relevant to the hypothesis. We ask the LLM to select a representative set of inputs to be used to test the hypothesis, including some counterexamples. It is up to the LLM's discretion to:

- call the *fuzzer* as often as deemed necessary
- include its own inputs (e.g. if the fuzzing results are not the the LLM's satisfaction)
- decide which inputs to submit

As with the other nodes, we ask the LLM to be explicit and verbose in its reasoning. When the LLM decides on a set of suitable inputs, it submits them as a structured list of strings which is then evaluated by the oracle and returned to the *learner*.

4.4 Fandango Fuzzer

In order to reliably generate syntactically valid inputs for our program under test and still be able to shape these inputs according to the hypothesis, we employ the *Fandango* [34] fuzzer. First we pass the grammar as well as the generated formal constraints to *Fandango* to then call the `fuzz` function to generate the desired amount of inputs as specified by the generator. We then repeat the fuzzing process with one of the constraints negated by simply surrounding the corresponding string with parentheses and prefixing the '`not`' keyword. We repeat this process for every constraint in our list. Unfortunately, *Fandango* often struggles with constraints that are negated in this fashion. As a fallback, we have slightly modified the system prompt of the *constraint formalizer* to instead negate formal constraints, avoiding simply prefixing '`not`' as this would have already failed. Finally, the fuzzer node returns a list of fuzzed inputs to the generator.

4.5 Example

We conclude the presentation of *HDDAgent* with a practical example. First, we introduce a context-free grammar for a simple calculator subject. For better readability of the grammar, we define the following symbols:

- Non-terminal symbols are encapsulated in angled brackets $\langle\ \rangle$

- Terminal symbols are represented in plain text
- alternative productions are separated by a vertical bar |
- ⟨`symbol`⟩$^?$ or `symbol`$^?$ indicates that the symbol is optional
- ⟨`symbol`⟩$^+$ or `symbol`$^+$ means that the symbol may be expanded once or more

The grammar for the calculator subject can now be written as follows:

$$
\begin{aligned}
\langle \texttt{start} \rangle &\rightarrow \langle \texttt{arith\_expr} \rangle \\
\langle \texttt{arith\_expr} \rangle &\rightarrow \langle \texttt{function} \rangle (\langle \texttt{number} \rangle) \\
\langle \texttt{function} \rangle &\rightarrow \texttt{sqrt} \mid \texttt{sin} \mid \texttt{cos} \mid \texttt{tan} \\
\langle \texttt{number} \rangle &\rightarrow -^? \langle \texttt{one\_nine} \rangle \langle \texttt{digits} \rangle^? \langle \texttt{frac} \rangle^? \\
\langle \texttt{frac} \rangle &\rightarrow . \langle \texttt{digits} \rangle \\
\langle \texttt{digits} \rangle &\rightarrow \langle \texttt{digit} \rangle^+ \\
\langle \texttt{one\_nine} \rangle &\rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\langle \texttt{digit} \rangle &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

The learner invents the following hypothesis:

"The program fails when the argument to the 'sqrt' function is a negative number."

Due to the simplicity of our example, the LLM immediately identified the correct failure conditions but it will still have to collect further evidence to be certain. The hypothesis is thus passed to the constraint generator which derives the following constraints:

- The function must be 'sqrt'.
- The number must be negative.

The constraint formalizer then translates them into the following Fandango constraints:

- `<function> == "sqrt"`
- `str(<number>).startswith("-")`

The generator then calls the fuzzer to generate new inputs, producing 10 inputs that satisfy both the given grammar and the constraints above. For each constraint, we also generate 10 inputs where the respective constraint is negated for a total of 30 inputs. The LLM then analyses the generated inputs and identifies those that match the constraints as well as the counterexamples. After deciding, that there is no need for additional inputs, it then selects the inputs that will be passed to the oracle. The amount of inputs to select is up to the LLM's discretion. In our example, the LLM decided on 5 inputs, among which 3 satisfy both constraints:

- `sqrt(-2876)`
- `sqrt(-424)`
- `sqrt(-6)`

as well as two counterexamples, one for each negated constraint:

- `cos(-84)`
- `sqrt(33)`

The oracle identifies `sqrt(-2876)`, `sqrt(-424)` and `sqrt(-6)` as failing, while `cos(-84)` and `sqrt(33)` pass. The results from the oracle are then passed to the learner node which correctly determines that the experiments align with the hypothesis. Finally, the LLM decides that it cannot refine its hypothesis any further and therefore returns the final diagnosis:

```
The program fails when the 'sqrt' function is provided with
a negative argument, as square roots of negative numbers are
undefined in the real number system. This is consistent with all
failing test cases (e.g., sqrt(-900), sqrt(-2876), sqrt(-424),
sqrt(-6)) and the passing cases (e.g., cos(-84), sqrt(33)).
Other functions like 'cos' correctly handle negative arguments,
confirming the issue is specific to 'sqrt' and negative inputs.
```

As demonstrated in this example, the LLM has a tendency to add additional explanations to its diagnosis, for instance by providing the mathematical context of the failure or by referencing test results. While one might argue that this behavior is outside the scope of automatically diagnosing program failures, research [17, 19] in other areas of automated software engineering has shown that supplementary information is often considered to be helpful by software developers.

5 Methodology

For our evaluation, we address the following research questions:

RQ1 How effective is *HDDAgent* at identifying the exact failure conditions of programs compared to *Avicenna*?

RQ2 How effectively do LLMs reason about failure conditions and to what extent does it align with scientific debugging practices?

Since our approach outputs diagnoses in natural language, we measure the effectiveness by first classifying the diagnoses produced by *Avicenna* and *HDDAgent* and then comparing the amount of diagnoses that accurately describe the true failure conditions for each approach per subject. We explain our classification in more detail in subsection 5.2.

To answer **RQ2** we perform a qualitative analysis of the reasoning content produced by the LLM. To that end we select one run representing each category we defined in subsection 5.2 as well as a run that was aborted due to reaching the limit for node transitions.

5.1 Evaluation Setup

In order to evaluate the effectiveness of our approach, we use a selection of subject programs of varying degrees of complexity. Calculator, middle and markup are synthetic subjects. For real-world examples we use Cookiecutter with 3 samples and Pysnooper with 2 samples in addition to 2 SWE-Bench subjects, Requests and Astropy.

Benchmark Subjects

From the debugging-benchmark repository [9] we use the following subjects:
Calculator is the same program we presented in section 4.5. It fails when the `sqrt` function is provided with a negative number as argument.
Middle takes three positive integers $x, y, z \in \mathbb{Z}^+$ as arguments and returns the median, however, the implementation is flawed and returns a wrong result when $y < x < z$, resulting in a failing oracle result.
Markup is the final synthetic subject and removes HTML tags from a provided input string. For inputs, where text outside the tags contains ¨,ˆ, or a leading or trailing space, the program fails.
Cookiecutter is a command-line utility that creates projects from templates. We use the following 3 samples which stem from real GitHub issues:

1. The program fails when multiple pre- or post-hooks are provided
2. The program fails when a list is provided for a key instead of a single value.
3. The program fails when a post-hook provides a non-zero return code.

Pysnooper is a debugging-tool for python that reports on line execution and changes of local variables. The 2 buggy samples we use are:

1. The program fails when using the `custom_repr` argument.

2. The program fails when the `output` argument is used.

Requests and **Astropy** are two samples found in SWE-bench [14]. Since SWE-bench is intended to be used as a benchmark to evaluate the ability of LLMs to resolve real-world GitHub issues, we had to manually construct the test oracles for both samples. The *Astropy* sample we chose contains a bug in the `Latitude` class. When initialized with an angle not in the range between $-90°$ and $90°$, a `ValueError` exception will be thrown. Due to rounding errors, this behavior leads to an error When initializing `Latitude` with an angle of $\frac{\pi}{2}$rad or $-\frac{\pi}{2}$rad using 32-bit floating point precision. This behavior is unintentional and has been fixed in newer versions of *Astropy*. To construct the oracle, we simply catch any exceptions thrown by `Latitude` and return a failing oracle result whenever the angle is within 5 decimal places of $-90°$ or $90°$ respectively.
Our chosen *Requests* sample fails when `requests.get` is used to retrieve or follow an URL that contains a % character. In order to prevent issuing excessive amounts of web requests to URLs that may or may not exist, we opt to use a surrogate oracle that returns a failing oracle result whenever it is passed a string containing the % character, returning a passing result otherwise. For our grammar, we used an ANTLR-compatible URL grammar [8] as a basis and adapted it for compatibility with Avicenna and Fandango respectively.

5.2 Diagnosis Classification

We define the *input space* of a program as the set of all syntactically correct inputs as per the programs input grammar. For our purposes we additionally assume that the input space does not contain inputs that lead to spurious failures. Diagnoses are descriptions of input features that lead to program failure. As such, a diagnosis can be understood as a subset of the program's *input space*. For inputs that lie within that subset, we expect the program to fail while inputs outside of that subset are expected to pass. It is important to note that the diagnoses produced by Avicenna or our approach are not guaranteed to perfectly reflect the program's behavior. We therefore define the *ground truth diagnosis* to be a perfect description of a programs failure conditions that correctly predicts the programs behavior for the entire input space. We can now compare the diagnoses produced by Avicenna and our approach to the ground truth diagnosis and classify them into the following categories as illustrated in figure 4:

exact - The diagnosis and the ground truth describe the same subset of the input space. Therefore, the program fails if and only if the input matches the conditions described by the diagnosis. There are no false positives or false negatives and the precision and recall should both be 1.
overspecific - The diagnosis is a non-empty, proper subset of the true failure conditions.

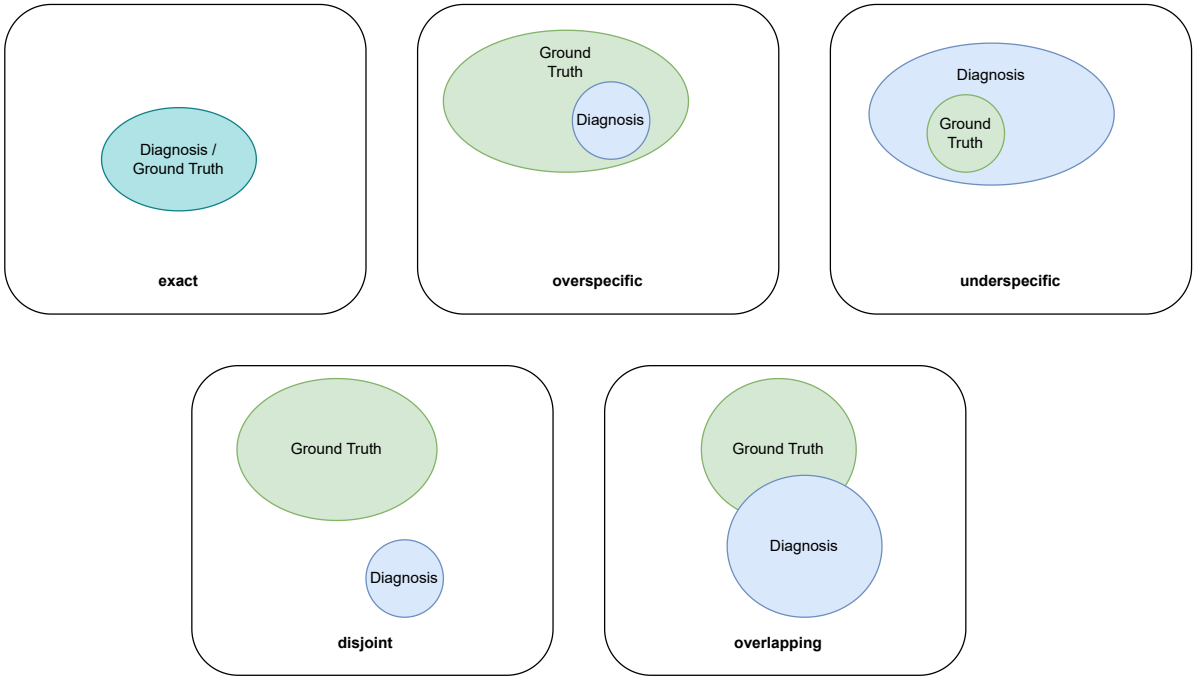

Figure 4: Classification of diagnoses when compared against the ground truth.

All inputs that should fail as per the diagnosis do in fact fail, but there are failing inputs that should not fail according to the diagnosis. There are no false positives and the precision should equal 1, but there are false negatives and the recall should be less than 1.
underspecific - The diagnosis is a proper superset of the true failure conditions. Every failing input is covered by the diagnosis but there are passing inputs that should fail according to the diagnosis. There are false positives, leading to a precision that is less than 1, however there are no false negatives. Therefore, the recall will be equal to 1.
disjoint - The diagnosis and the ground truth are disjoint. This includes the special case where the diagnosis describes the empty set. The predictions of the diagnosis are incorrect for all inputs. The expected precision and recall are 0.
overlapping - The diagnosis is neither a subset nor a superset of the ground truth, however, both sets are not disjoint. The diagnosis makes correct predictions for some inputs, but there are passing inputs that are predicted to fail and failing inputs that are predicted to pass as per the diagnosis. Both, precision and recall should be in the interval $(0, 1)$.

To answer **RQ1**, we run Avicenna and our approach respectively for 30 runs per subject. We then compare the generated diagnoses to the respective ground truth, classifying the result into the categories we just defined. In cases where Avicenna outputs multiple diagnoses with the same measured precision and recall, we select the first diagnosis from the list for our comparison. For calculator, middle and markup we derive the ground truth diagnosis from the benchmark harness function and the oracle found in the debugging-benchmark repository. For Cookiecutter and Pysnooper we use the

explanations provided in the debugging-benchmark [9] repository. For Requests and Astropy we use the corresponding GitHub problem statements as the ground truth.

5.3 Implementation and Runtime Environment

We implemented our approach in a *uv*-managed virtual Python 3.12.11 environment. For our LLM, we chose Qwen3-30B-A3B-Thinking-2507-FP8 due to its state-of-the-art performance among open-source models with similar parameter size. To run Qwen3, we use vLLM's OpenAI-compatible server. The workflow is implemented using LangGraph, an orchestration framework for building agents and agentic workflows. We used the default `GRAPH_RECURSION_LIMIT` of 25, meaning the run is aborted when exceeding 25 steps in order to prevent excessive runtime. We compared our approach against the Avicenna implementation found in the dbg repository [4]. Additional implementation details such as the exact package versions used can be found in our GitLab repository: https://gitlab.informatik.hu-berlin.de/passonda/hypothesis-driven-debugging-with-large-language-models.
For Avicenna, we ran Cookiecutter.3 on an AMD Ryzen 7 9800X3D processor and Requests on an Asus ESC4000 server equipped with 2 Xeon 6354 CPUs. All other experiments were conducted on a Dell R740 server equipped with 2 Intel Xeon 6134 CPUs and 3 NVIDIA A100 80GB PCIe GPU, however, we only used a single GPU for LLM inference.

6 Evaluation

6.1 RQ1: Comparing the Performance of Avicenna and HDDAgent

Table 2 contains the results from our benchmarks. Numbers in parentheses under the disjoint category indicate the number of runs that failed without returning a diagnosis. In

	Avicenna					HDDAgent				
Subject	exact	overspecific	underspecific	disjoint	overlapping	exact	overspecific	underspecific	disjoint	overlapping
Calculator	27	3	0	0	0	**30**	0	0	0	0
Middle	0	0	0	0	30	**3**	0	2	0 (1)	24
Markup	0	0	4	0	26	0	2	0	0	28
Cookiecutter.1	0	0	0	0	30	**30**	0	0	0	0
Cookiecutter.2	0	0	29	0	1	**29**	0	0	0 (1)	0
Cookiecutter.3	0	0	13	0	17	**22**	0	8	0	0
Pysnooper.1	0	0	23	0	7	**26**	4	0	0	0
Pysnooper.2	0	0	30	0	0	**11**	19	0	0	0
Astropy	0	10	4	0	16	**1**	14	7	2 (4)	2
Requests	0	0	0	0	30	**30**	0	0	0	0
Total	27	13	103	0	157	182	39	17	2(6)	54

Table 2: Performance of Avicenna and our approach.

our experiments, our approach managed to generate more exact diagnoses than Avicenna for every subject except *Markup*, where neither approach was able to find a diagnosis equivalent to the ground truth. Notably, not a single Avicenna diagnosis was disjoint with the ground truth, while our approach returned 2 disjoint diagnoses in addition to 6 cases of no diagnosis being returned.

Looking at the generated diagnoses in more detail, for **Calculator**, all 3 overspecific diagnoses that Avicenna produced contain a wrong constraint for the ⟨`number`⟩ element, leading to the prediction of no failure for some negative numbers when used as arguments for the `sqrt` function.

For **Middle**, the most common Avicenna diagnosis was that the program fails when x is greater than 1 and z is not the smallest of the 3 numbers. For the other diagnoses, Avicenna learned some seemingly arbitrary boundaries (e.g. $z > 4, y < 67$). Our approach struggled with identifying the exact failure conditions as well, often finding some coincidental property of the initial inputs, e.g. all parameters being single-digit or the inputs satisfying $x + z \neq 2y$ as the cause for failure.

All 4 underspecific Avicenna diagnoses for **Markup** were identical with the failure condition being the existence of a ⟨`string`⟩ non-terminal in the parse tree, which encompasses all failing and passing inputs. For the overlapping category, the absence of empty ⟨`str`⟩ containers was the most common diagnosis from Avicenna. Our approach identified the presence of double quote characters (`"`) as problematic in every diagnosis, sometimes in addition to other special characters, but not all occurrences of double quotes lead to failure. Only in 2 instances did our approach produce a diagnosis without false positives, once managing to correctly identify the presence of double quotes outside HTML-tags as failure causing and in the other instance describing the special case where the input starts and ends with double quotes.

For our **Cookiecutter.1** sample, Avicenna identified a variety of seemingly arbitrary

characters as failure-inducing where none of the identified input features were exclusive to either failing or passing inputs.

In all but one diagnosis for **Cookiecutter.2**, the input features identified by Avicenna are present for all passing and failing inputs making the diagnoses trivially underspecific. For the overlapping diagnosis, Avicenna concluded that the program fails when every integer in the input is less than or equal to 973158, which of course may or may not be the case for any passing or failing input.

All underspecific diagnoses for **Cookiecutter.3** describe the existence of an integer value greater than 0 as failure-inducing, while the remaining overlapping diagnoses consist of seemingly arbitrary constraints for various input features that don't relate to the failure, e.g. `str.to.int`$(\langle int \rangle) < 424242$ or `str.to.int`$(\langle day \rangle) > 1$. In all instances where our approach failed to generate an exact diagnosis, it instead concluded that both, `pre:exit` and `post:exit` hooks with non-zero return codes lead to failure, when in fact this is only the case for `post:exit` hooks.

For **Pysnooper.1**, Avicenna produced identical diagnoses within the underspecific and overlapping categories respectively, with the underspecific diagnosis being the existence of the mandatory ⟨`prefix`⟩ symbol in the derivation tree and the overlapping diagnosis attributing the failure to the presence of a space character as a separator. In two cases where our approach did not produce an exact diagnosis, it identified the presence of the `-c` tag (representing the ⟨`custom_repr`⟩ option) with non-matching `t_function` and `p_function` arguments as cause for failure while the other two overspecific diagnoses predict failure for `-c` not followed by a space and `-c` followed by an equal sign respectively.

All Avicenna diagnoses for **Pysnooper.2** attribute the failure to the presence of at least one ASCII character, which is a property that any failing or passing input fulfills. Our approach correctly identified the `-o` option as failure-inducing, however, it often included additional constraints such as the presence of a dot in the ⟨`path`⟩ or the lack of a space after the `-o` in the diagnosis.

For **Astropy**, Avicenna correctly identified an angle of $\frac{\pi}{2}$rad as failure-inducing, however this diagnosis does not generalize to angles of $-\frac{\pi}{2}$rad. The overlapping diagnoses are lacking the requirement of the unit being given in radians while the underspecific diagnoses require the presence of the constant π as well as a fraction with a 2 in the denominator. Our approach generated diagnoses for every category of our classification system. For the overspecific category, our approach omitted the case where the angle equals $-\frac{\pi}{2}$rad, while the underspecific diagnoses contained the requirement of a constant, a fraction or both as failure-inducing features. Both overlapping diagnoses predict failure when the angle is exactly $\frac{\pi}{2}$ but don't require the unit to be radians. Curiously, there are two instances where the LLM insisted that there are no failure conditions.

In all diagnoses for **Requests**, Avicenna attributes the failure to seemingly arbitrary input features such as `str.to.int`$(\langle port \rangle) \leq 780$ which do not relate to the actual failure conditions.

6.2 RQ2: Qualitative Analysis of LLM Reasoning

In the previous section we focused on the diagnoses produced by Avicenna and our approach respectively. As can be seen in Table 2, our approach produced diagnoses for every category for the **Astropy** sample. From each of these runs we select one for each category and summarize the `reasoning_content` that vLLM returns alongside the LLM response.

6.2.1 Case: Exact Diagnosis

Starting with the run that ended up producing an exact diagnosis, the LLM begins by summarizing the instructions given in the learner prompt.

```
Okay, let's tackle this debugging problem. So, the goal is to find
a hypothesis that explains why the program fails for certain inputs
but passes for others. The program takes inputs in a specific
grammar, and the oracle tells us if it's passing or failing.
```

The LLM then recalls the initial inputs alongside their corresponding oracle results, matching the first step of scientific debugging as outlined in section 2.1 which is to observe the failure of the program. Continuing onward, the LLM looks for patterns in the initial inputs while referencing the provided input grammar. It notices that the only failing input is also the only input to include the constant `np.pi` and formulates its first candidate for a hypothesis:

```
Wait, "np.pi" is a constant in the grammar. So maybe the program
has a problem when the angle is expressed using the constant np.pi.
Let's check the other inputs. The input "np.pi/2" uses the constant,
but others don't. The passing inputs don't use any constants. So
maybe the hypothesis is that the program fails when the angle is
expressed using the constant np.pi.
```

The LLM then starts scrutinizing its candidate hypothesis, noting that `np.pi` is a valid constant according to the grammar and that `np.e` is another possible constant, although it is missing from the initial inputs. While acknowledging that the program is to be considered a black-box, it then starts making assumptions about its internal handling of the pi constant like the program not correctly converting `np.pi` into a number. After considering the possible internal reasons for the failure, the learner then returns to analyzing patterns in the initial inputs.

```
Passing:
- 45,'deg' → angle=45 (int), unit=deg
- 3/4,'rad' → angle=3/4 (fraction), unit=rad
- 22/7,'rad' → angle=22/7 (fraction), unit=rad
- 3.14159,'rad' → angle=3.14159 (float), unit=rad
- 0,'rad' → angle=0 (int), unit=rad
Failing:
-np.pi/2,'rad' → angle=np.pi/2 (const expression), unit=rad
```

The LLM then invents a variety of alternate hypothesis but quickly abandons them. It is here however, that we detect a potential flaw in the LLMs reasoning.

```
Wait, but the grammar also allows 'np.e', so if the program failed
on 'np.e', that would suggest it's a general issue with constants.
But since we don't have data on 'np.e', the hypothesis should be
specific to 'np.pi' based on the given data.
```

It seems that the LLM is too fixated on the initial inputs at this stage and seems to avoid a more generalized hypothesis simply because it has no direct evidence that `np.e` might be a related to the programs failure. We would hope for the LLM to work towards disproving potential alternative hypotheses as well instead of just proving the most plausible one. In the final reasoning stage, the LLM experiments has decided to move forward with its initial hypothesis candidate and starts experimenting with different phrasings. After verifying that the hypothesis is concise enough and relevant to the input features, it then submits the hypothesis:

```
The program fails when the angle contains the constant 'np.pi'.
```

As we described in section 4, the hypothesis generated by the learner is then sent to the constraint generator which extracts the natural language constraint "The angle must contain the constant 'np.pi'." which is then formalized into the Fandango constraint `"np.pi" in str(`$\langle angle \rangle$`)`.

Next, the generator node is called and starts reviewing its task description before calling the fuzzer tool. The fuzzer generated 10 inputs that fulfill the given constraint as well as 10 inputs for the negated constraint. The generator node analyzes the features of the fuzzed inputs while considering the grammar and the given constraint. It identifies the positive and negative examples among the inputs and, after deeming the inputs to be sufficient, selects 5 of each to then submit them to the oracle, which evaluates each input and forwards the result to the learner.

The learner LLM recalls the initial failing input and notices that all newly generated inputs pass causing it to reject the hypothesis:

```
Okay, let's see. The initial inputs had "np.pi/2,'rad'" as failing,
but the new test cases with various np.pi expressions all passed.
Hmm, so my first hypothesis was that the program fails when the
angle contains 'np.pi', but the new tests show that even with np.pi
in different forms, it's passing. So that hypothesis must be wrong.
```

The LLM once again analyzes the features of the initial failing input while referencing the grammar. It suspects the issue to be related to fractions. Noticing that there were passing inputs containing fractions but none with a 2 in the denominator, the LLM invents a new candidate hypothesis:

```
Wait, but the initial input "np.pi/2" failed, and the new test
"np.pi/4" passed. So maybe the denominator has to be 2? Let's think.
If the denominator is 2, it fails. But why would denominator 2 be
a problem? Maybe the program has a special case where it handles
denominator 2 differently, but that's not clear.
```

Unable to come up with a plausible reason for why a fraction with a denominator of 2 might be failure-inducing, the LLM continues formulating alternatives but quickly abandons each of them in light of the oracle results. Trying again to understand the reason behind the failure, the LLM converts the inputs to degrees and notices a pattern:

```
Wait, but the input "0,'rad'" passed, which is 0 radians.
"3.14159,'rad'" is approximately pi radians, which should pass.
So maybe the program has a bug when the angle is exactly pi/2
radians, i.e., 90 degrees. But why would that be a problem? Maybe
the program has a special case for 90 degrees that's broken.
```

Remarkably, while continuing its attempt to understand the failure, the LLM was able to find an explanation that is close to the actual root cause, which is the occurrence of a rounding error, albeit when converting pi from 32-bit precision to 64-bit precision and not when using trigonometric functions as the LLM suspected.

```
But why would the program have a bug at 90 degrees? Maybe it's a
rounding error or a special case that's not handled. For example,
if the program is trying to calculate sine or cosine of the
angle, and at 90 degrees, it's encountering a division by zero
or something. But the problem is that the program is failing on
input parsing, not on computation. Wait, the oracle result is about
the program failing, which could be a crash or an error during
processing.
```

Another interesting property we can observe from this line of reasoning is that the

LLM wrongfully assumed the failure to be caused by a parsing error, yet it was able to self-correct its assumption. Despite having found an almost correct understanding of the root cause for failure, the LLM returns to its previous explanation that a fraction with a denominator of 2 causes the failure and submits it instead.

```
The program fails when the denominator of the angle fraction is
exactly 2.
```

After constraint extraction and formalization, the resulting Fandango constraint is ⟨nonzero⟩== "2", as the denominator of the fraction is represented by the ⟨nonzero⟩ non-terminal which can be derived to any digit between 1 and 9. Like in the previous iteration, the generator calls the fuzzer and selects inputs that satisfy the constraint as well as counterexamples that were generated using the negated constraint. This time the generator submits 3 positive and 3 negative examples to the oracle. In the learner node, the LLM recalls the previous hypotheses and concludes from the oracle results that neither hypothesis holds. As one of the new failing inputs is `-np.pi/2, 'rad'`, the LLM is able to refine its first hypothesis.

```
Wait, the initial failing input was "np.pi/2,'rad'". Then the next
test input "-np.pi/2,'rad'" also failed. So maybe the issue is when
the angle is exactly 'np.pi' over 2, regardless of the sign? But
"np.pi/4,'arcsec'" passed. So it's specifically denominator 2 and
numerator 'np.pi'?
```

It then further refines the hypothesis, by including the unit as well:

```
Wait, the unit matters here. The initial failing input was in 'rad',
and the new failing input "-np.pi/2,'rad'" is also in 'rad'. The
other inputs with denominator 2 but in 'deg' passed. So maybe the
failure is when the angle is (np.pi / 2) in radians. But why would
that be?
```

The LLM then verifies, that the now refined hypothesis is consistent with all previous test inputs. It is also able to rule out `np.e` playing a role in the programs failure as the most recent batch of tests included a passing result for an input containing the constant.

```
Now, the next step is to check if this hypothesis can be further
refined. But given the test results, this seems to fit all the data.
Let's see if there's any input that should fail but doesn't, or
vice versa.
Looking at the test inputs provided after the second hypothesis:
- "-np.pi/2,'rad'" → FAILING (matches hypothesis)
- "21*-236/2,'deg'" → PASSING (doesn't match the hypothesis's angle
and unit, so correct)
- Others passed as expected.
So the hypothesis holds. Now, can it be made more concise? The
hypothesis is already specific: angle is ±np.pi/2 and unit is 'rad'.
There's no need for further refinement. So this should be the final
diagnosis.
Wait, but the user's previous hypothesis was about denominator 2,
but now we've refined it to include the specific numerator and unit.
So the correct structured hypothesis should state that.
So the final hypothesis is: The program fails when the angle
is exactly ±np.pi/2 (i.e., the angle expression is "np.pi/2" or
"-np.pi/2") and the unit is "rad".
```

While the diagnosis ended up to be correct, the LLM seemed overly eager to submit a diagnosis rather than spend another iteration verifying the hypothesis. That being said, the reasoning demonstrated by the LLM was overall sound, albeit rather unstructured. Rather than a continuous chain of thought, the LLMs reasoning seems to resemble several chain fragments strung together as the LLM was frequently and rapidly jumping from one idea to another.

6.2.2 Case: Overspecific Diagnosis

In the run that produced an overspecific diagnosis, the LLM immediately starts analyzing the initial inputs without summarizing its instructions first. Like in the previous run, the LLM notices that the constant `np.pi` is only present in the failing input and searches for possible explanations for this behavior before formulating its first candidate hypothesis:

```
So the hypothesis could be: The program fails when the angle is
specified using the constant "np.pi" (e.g., "np.pi/2"), but passes
for all other valid numerical inputs.
```

The LLM then briefly considers a few alternative explanations before returning to its original candidate. Using the candidate hypothesis, it then makes predictions on how the program should behave for future inputs.

```
Now, to confirm this, we need to test inputs with "np.pi" in
different positions. For example:
- "np.pi,'rad'" → should fail
- "2*np.pi,'rad'" → should fail (since it contains "np.pi")
- "np.pi*2,'rad'" → should fail
- "3.14,'rad'" → should pass (as it's a numerical value)
```

In the following, the LLM seems to be confused about the grammar and the fact that inputs can fail despite being grammatically valid. It then briefly questions whether there are any other viable hypotheses but just like in our previous example, it ultimately settles for the first candidate, which it submits after verifying its validity.

```
The program fails when the angle input contains the string 'np.pi'
because it does not recognize this as a valid numerical constant,
causing a parsing or evaluation error.
```

Despite being instructed to keep hypotheses relevant to the input features, the LLM included assumptions about the programs internals. The constraint generator and constraint formalizer nodes ultimately produce the following formal constraints: `str(`$\langle angle \rangle$`)== "np.pi"` and `str(`$\langle unit \rangle$`)in ["deg", "rad", "arcmin", "arcsec"]`. Note, that the second constraint is redundant as it follows directly from the grammar. Also of note is, that Fandango seems to struggle with this particular set of constraints as the fuzzer kept throwing exceptions even after 3 retry attempts. Despite these issues, Fandango was able to generate inputs satisfying the constraints when called by the generator node, however it was unable to produce any inputs for the negated constraints. Not seeing any counterexamples in the test inputs, the generator ended up calling the fuzzer 3 times. The LLM expresses some confusion as to why the fuzzer did not create any counterexamples, leading it question whether it correctly interpreted the prompt. In the end however it decided to create its own counterexamples.

```
But the fuzzer, when called with the constraints, shouldn't
generate counterexamples. So the user has to generate them
manually. Given that, the user should take some inputs from the
fuzzer's output (positive examples) and create some counterexamples
by replacing 'np.pi' with a valid number. For example, take
"np.pi/3,'deg'" and change it to "3/3,'deg'" → "1,'deg'", which
is valid grammar and unit, but angle doesn't have 'np.pi'.
```

Finally, the generator submits 3 positive and 3 negative examples to the oracle which passes the results to the learner. Here, the LLM immediately notices that its hypothesis has been disproved and considers a series of alternative hypothesis and ultimately ends up extending its original hypothesis to also require the unit to be given in radians.

```
The program fails when the angle input contains 'np.pi' AND the
unit is 'rad', because the program attempts to parse the angle as
a numerical expression for radians but cannot recognize 'np.pi' as
a valid constant, whereas other units (like 'deg') do not require
numerical evaluation of the angle.
```

Once again there were issues generating valid Fandango constraints from the hypothesis. After 3 retries the the generated formal constraints were ⟨*angle*⟩ `== "np.pi"` and ⟨*unit*⟩ `== "rad"`. Despite being provided with parts of the Fandango specification and some additional examples, the constraint formalizer did not cast the non-terminals to a string, causing the fuzzer to throw an exception when attempting to validate the constraints. Moving on however, the generator ended up not calling the fuzzer anyway and instead reused some of earlier fuzzer outputs and submitting them. From the oracle results, the learner then quickly notices a passing result for the input `np.pi,'rad'`, leading it to conclude that its hypothesis has been refuted. As per usual, the LLM then jumps between many different hypotheses oftentimes repeating itself. Notably, the LLM does not seem to consider inventing hypotheses for the primary purpose of generating new tests as a possibility.

```
Wait, the latest test didn't include "np.pi/2,'deg'". If we had
tested that, we could see if it fails in 'deg' as well. But since
we don't have that data, we can't.
```

Instead the LLM appears to be focused on finding an explanation within the current iteration, despite acknowledging the lack of data as a limiting factor in finding suitable hypotheses. Ultimately, the LLM decides to submit its final diagnosis alongside a wrong, yet lengthy assumption of the root cause for the failure without any further testing.

```
The program fails specifically when the angle input is 'np.pi/2'
with the unit 'rad'. This is because the program's internal
handling of the angle value 'np.pi/2' in radians triggers a parsing
or evaluation error, while all other combinations involving 'np.pi'
(e.g., 'np.pi', 'np.pi/3', 'np.pi/7', '-np.pi') with any unit pass.
The failure is not due to the presence of 'np.pi' alone or the
'rad' unit alone, but the specific combination of 'np.pi/2' in
radians. This indicates a narrow edge case in the program's angle
processing logic for this exact input pattern.
```

6.2.3 Case: Underspecific Diagnosis

Beginning the analysis of the run that lead to an underspecific diagnosis, we find that the reasoning of the learner is overall very similar to the previous runs. After recalling the

instructions, the inputs, and the grammar, the LLM tries to find plausible root causes for the programs behavior before stating its first hypothesis candidate.

```
But the hypothesis has to be based on input features. So the
presence of "np.pi" in the input might be the key. Let's see: all
passing inputs don't have "np.pi" or "np.e", while the failing one
does. So the hypothesis could be that the program fails when the
angle contains "np.pi".
```

Like in both previous runs, the LLM relates its first candidate hypothesis to the presence of the `np.pi` constant and despite briefly considering possible alternatives, the LLM decides to move forward with the initial candidate this time as well.

```
The program fails when the angle input contains the constant
'np.pi'.
```

Moving forward, the LLM derives the same initial constraint as in the run that produced the exact diagnosis, namely `"np.pi" in str`($\langle angle \rangle$). Without any validation issues, the execution enters the generator node which as we would expect, ends up calling the fuzzer tool after planing its workflow. Without any issues, the fuzzer returns 10 inputs fulfilling the constraint as well as 9 counterexamples. After selecting 3 inputs each for a total of 6 inputs submitted, we once again return to the learner node where, once again, the LLM quickly realizes the need to refine its hypothesis.

```
The first hypothesis I came up with was that the program fails
when the angle input contains 'np.pi'. But then, after testing
the new inputs, all the ones with 'np.pi' (like "np.pi,'rad'",
"-np.pi/5,'rad'", "np.pi/1,'deg'") are passing. Wait, that's a
problem. Let me go back to the initial data. The initial failing
case was "np.pi/2,'rad'". The hypothesis was that any input with
'np.pi' causes failure. But when they tested "np.pi,'rad'", it
passed. So the hypothesis is contradicted because the failing
case was "np.pi/2", but other 'np.pi' cases are passing. So the
hypothesis needs to be refined.
```

Trying to find an alternative explanation, the LLM focuses on the 2 in the denominator of the initial failing input `np.pi/2,'rad'`. Curiously, the LLM considers the angle being $\frac{\pi}{2} rad$ as a possible hypothesis but seems to find the case that the program is hard-coded to fail when `/2` is present in the input to be more plausible.

```
So the correct hypothesis must be more specific. Maybe it's when
the angle is exactly half of pi (np.pi/2) in radians. Let's
check the other inputs. The initial passing inputs include
"3.14159,'rad'" which is approximately pi, but that's a float,
not using np.pi. So maybe the problem is with the exact string
"np.pi/2" rather than the value. Wait, but "np.pi/2" is a valid
input according to the grammar. The program might be trying to
parse "np.pi/2" as a string and not evaluate it, leading to a
failure. But why would "np.pi/1" work? Because "np.pi/1" is the
same as "np.pi", but written as a fraction. Wait, maybe the program
expects the fraction to have a denominator that's a single digit?
Or maybe it's a typo in the code where it's checking for "/2"
specifically.
```

Once again the LLM seems to have an aversion to continuing the loop and chooses to submit its diagnosis instead.

```
The program fails when the angle input contains the substring
'/2', regardless of the unit or other input components. This is a
string-matching bug in the program's input validation logic, where
any occurrence of '/2' in the angle string triggers a failure.
```

6.2.4 Case: Disjoint Diagnosis

As we have by now explored the common patterns found in the LLMs reasoning content in great detail, we will abbreviate passages that show patterns similar to the ones we already explored. In the case of the run that lead to a disjoint diagnosis, the learners initial hypothesis matched that of the previous runs with no remarkable findings within the reasoning.

```
The program fails when the angle contains the constant 'np.pi'
because it does not correctly evaluate this symbolic constant
to its numerical value (approximately 3.14159), while other
valid numerical inputs (e.g., '3.14159', '22/7', '3/4') pass
successfully.
```

Like in section 6.2.2 we observe difficulties regarding the constraint generation, however this time, the LLM seems to have generated valid formal constraints with `str(`$\langle angle \rangle$`) == "np.pi"` and `str(`$\langle unit \rangle$`) in ["deg", "rad", "arcmin", "arcsec"]`. Still in the subsequent fuzzer call by the generator node, Fandango was unable to generate any inputs, returning an empty list to the generator. After another failed fuzzing attempt, the LLM concludes that there must be an error with the fuzzer and begins constructing inputs on its own.

```
Hmm. Since the fuzzer isn't working as expected, maybe I should
manually create the inputs. The valid inputs should be:
np.pi,deg
np.pi,rad
np.pi,arcmin
np.pi,arcsec
And the counterexamples would be, for example, 3.14,deg (which
should pass), or np.pi, invalid_unit (which should fail).
```

The generator continues refining its input candidates for a bit and the submits the following inputs: `'np.pi,deg'`, `'np.pi,rad'`, `'3.14,deg'`, `'np.pi,invalid'`

Despite the last input being grammatically invalid, the learner evaluates the oracle results as usual and submits a refined hypothesis.

```
The program fails when the angle is expressed as a fraction
(contains '/') where the numerator is the symbolic constant 'np.pi'
(e.g., 'np.pi/2'), but passes when the angle is a single symbolic
constant (e.g., 'np.pi') or a numeric fraction (e.g., '3/4').
```

Our approach once again struggles with constraint generation, however, after two retries it finds the following constraint that validates without errors:

$$\texttt{str}(\langle angle \rangle) \texttt{ == "np.pi/" + str}(\langle nonzero \rangle)$$

Unfortunately the generator ends up not calling the fuzzer, assuming it is still broken and instead submits its own inputs including some counterexamples. The LLM-generated inputs, however, are syntactically flawed. Furthermore, one of the submitted inputs was `np.pi/2,rad` (missing the single quotes around `rad`), which causes Astropy to throw a different exception than the one we use to identify the proper failure condition, resulting in the oracle identifying `np.pi/2,rad` as a passing input. The LLM actually notices the difference in syntax but wrongfully concludes that the initial inputs must have been flawed.

```
Wait, the initial input was "np.pi/2,'rad'", and the new test
input was "np.pi/2,rad" without the quotes around 'rad'? Or
maybe the comma placement. Wait, the grammar says <start>
::= <angle>","<unit>, so the input should be angle,unit. So
"np.pi/2,'rad'" would have the unit as 'rad' (with quotes), but
maybe in the new tests, they wrote 'rad' without quotes, but the
grammar requires quotes. Wait, the grammar says unit is 'deg' |
'rad' | etc., so the unit must be in quotes. So maybe the initial
failing input had a typo, like missing quotes, but the new tests
had correct formatting.
```

The learner then continues to express its confusion about the oracle results and questions the correctness of the grammar. In the end, the LLM concludes that the initial failing input was indeed wrong and submits a disjoint diagnosis.

```
The program does not have a failure condition related to the
input features involving 'np.pi' in the angle. The initial oracle
result indicating failure for 'np.pi/2,'rad'' was likely due to a
transient or contextual error (e.g., input formatting inconsistency,
environment issue), as all subsequent tests with identical or
similar inputs (e.g., 'np.pi/2,rad', 'np.pi/1,deg', 'np.pi,rad')
consistently pass. The grammar and valid inputs (e.g., '3/4,'rad'',
'22/7,'rad'') confirm that fractional angles with numeric values
are handled correctly, and the symbolic constant 'np.pi' is
processed correctly in both standalone and fractional forms. No
input feature pattern explains the initial failure, indicating it
was an anomaly rather than a program defect.
```

6.2.5 Case: Overlapping Diagnosis

Lastly, we evaluate a run that lead to an overlapping diagnosis. The learner begins by generating the same initial hypothesis as in the run we evaluated previously.

```
The program fails when the angle contains the constant 'np.pi'.
```

After constraint extraction and formalization the resulting Fandango constraint ended up being identical to that of the first run which produced an exact hypothesis:

`"np.pi" in str(`$\langle angle \rangle$`)`

With no errors, the fuzzer is able to generate 20 inputs out of which the generator selects 3 positive examples and 3 counterexamples. The learner immediately notices that the new oracle results contradict its first hypothesis.

```
Okay, let's see. I need to figure out why the program fails when
given "np.pi/2,'rad'" but passes on other inputs. The initial
hypothesis was that the program fails when the angle contains
'np.pi'. But the new test cases all have 'np.pi' in them, like
"np.pi/7,'deg'", "-np.pi,'rad'", "np.pi*0.4/4,'arcmin'", and they
all passed. So the original hypothesis is wrong.
```

Searching for alternative explanations the LLM identifies the number 2 in the denominator of the angle as a possibly failure-inducing.

```
So the failing input is the only one with "np.pi/2" in 'rad'. The
other inputs with 'np.pi' (like "np.pi/7") passed. So the key
difference is the denominator 2.
```

So far, this run showed a strong resemblance with the run that lead to an exact diagnosis, however instead of exploring the hypothesis about the denominator independently, the LLM decides to combine it with its first hypothesis.

```
So the refined hypothesis is: "The program fails when the angle is
exactly 'np.pi' divided by 2."
```

Furthermore the LLM decides that its newly refined hypothesis fits the existing data well enough to warrant forgoing additional testing and ends up submitting the diagnosis.

```
The program fails when the angle is exactly 'np.pi/2' regardless
of the unit. This is due to a specific handling of the division by
2 in the angle value involving the 'np.pi' constant, which causes
a failure condition in the program's processing logic. All other
inputs containing 'np.pi' with different denominators (e.g., '/7',
'/4') or different units pass, confirming the failure is specific
to 'np.pi/2'.
```

7 Discussion

Our approach managed to produce at least one exact diagnosis for 9 out of 10 subjects in our benchmark while *Avicenna* was only able to produce exact diagnoses for one subject. Furthermore, our approach was able to generate exact diagnoses consistently for the majority of benchmark subjects with over 60% of all diagnoses produced being exact compared to 9% for *Avicenna*.

To answer **RQ1** we therefore conclude that our approach performed significantly better at identifying the exact failure conditions of our benchmark subjects than *Avicenna*.

Through our qualitative analysis of LLM reasoning we were able to observe some of the limitations of our approach. The generation of new test inputs proved to be challenging for complex failure conditions as evidenced by a multitude of invalid constraints leading to fuzzing errors. Additionally, Fandango exhibits difficulties when dealing with negated constraints, hindering our ability to produce counterexamples to the generated hypotheses. Considering the lack of foresight the LLM exhibited when inventing hypotheses, being unable to reliably generate counterexamples poses a significant disadvantage for our approach. Our observations showed, that the LLM

Answering **RQ2**, we conclude that LLMs reason about failure conditions using loosely connected chain-of-thought fragments. While we were able to find a coarse structure guiding the LLM through each step of its task, the LLM seems unable to express long coherent thought and is instead rapidly jumping from one idea to the next. This behavior stands in stark contrast to the systematic design of the scientific debugging method.

7.1 Threats to Validity

7.1.1 Threats to Internal Validity

For our comparisons we used the most recent version of *Avicenna* [4] which is a re-implementation of the original approach. Our results show a performance slightly lower than expected when compared the original *Avicenna* [10] paper, however we were unable test both versions due to time constraints. Another threat to internal validity is the selection of initial inputs as our approach has proven to be sensitive to the test results available to the LLM.

The nondeterministic nature of LLMs and fuzzers introduces another potential threat to internal validity, though we argue that we alleviated this factor by repeating each experiment 30 times.

Lastly, both tools expect different grammar formats with slightly different features. While the grammars for each subject should be equivalent in that they describe the same input language, we cannot rule out the possibility of the grammar design affecting the input generation e.g. by inducing biases.

7.1.2 Threats to External Validity

When evaluating LLM-based tools, data leakage can be a potential threat to external validity. As we used a recent open-source LLM for our implementation, it is possible that information about the subjects used in our evaluation is included in the LLM's training data, which might lead to improved performance on these samples that does not generalize to future bugs.

Furthermore we conducted all our tests using a single LLM, namely Qwen3-30B-A3B-Thinking-2507-FP8, and can therefore not guarantee that our qualitative analysis of LLM reasoning is comparable across all current or future chain-of-thought LLMs.

8 Conclusion and Future Work

In this thesis we proposed *HDDAgent*, a tool that automatically identifies failure inducing input features. Drawing inspiration from *Avicenna*, we built an agentic workflow that uses large language models with strong reasoning capabilities to automatically generate, test and refine debugging hypotheses in order to explain program failures in respect to its input features. Our findings showed, that our approach outperformed *Avicenna* in all but one subject where neither tool was able to find an exact diagnosis.

We also identified some limitations of our approach such as the LLM lacking the foresight to invent hypotheses to gain insights into the program's behavior as well as inconsistencies with the input generation.

Future work could explore more autonomous approaches for the input generation in particular. Our approach follows a strictly linear pipeline that uses the current hypothesis to generate constraints which are then passed to the Fandango fuzzer. For the event that the fuzzer fails to generate inputs for the generated constraints, we implemented a simple retry loop in hope of the LLM eventually generating a working list of constraints due to its nondeterministic nature. With failed input generation being a major cause of flawed diagnoses, a more complex approach for input generation may be necessary. An agent might be able to debug failed fuzzer calls thus improving the reliability of the input generation component.

Another direction worth exploring is to improve the capabilities of the LLM itself. Our qualitative analysis showed that LLM reasoning is chaotic, rapidly jumping between ideas rather than systematically exploring one thought at a time like a human developer. With chain-of-thought being a capability that is added to LLMs in post-training, it could be feasible to fine-tune an LLM to reason in a more methodical way that more closely resembles the way humans approach scientific debugging.

Lastly, future work could test the feasibility of our approach on a larger and preferably more recent test suite to avoid data leakage and compare a variety of LLMs including commercial state of the art models.

Bibliography

[1] M. Adnan, Z. Xu, and C. C. N. Kuhn. Large Language Model Guided Self-Debugging Code Generation, Feb. 2025. URL http://arxiv.org/abs/2502.02928. arXiv:2502.02928 [cs].

[2] A. Alaboudi and T. D. Latoza. Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, San Francisco CA USA, Oct. 2023. ACM. ISBN 979-8-4007-0132-0. doi: 10.1145/3586183.3606781. URL https://dl.acm.org/doi/10.1145/3586183.3606781.

[3] Y. Bajpai, B. Chopra, P. Biyani, C. Aslan, D. Coleman, S. Gulwani, C. Parnin, A. Radhakrishna, and G. Soares. Let's Fix this Together: Conversational Debugging with GitHub Copilot. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–12, Liverpool, United Kingdom, Sept. 2024. IEEE. ISBN 979-8-3503-6613-6. doi: 10.1109/VL/HCC60511.2024.00011. URL https://ieeexplore.ieee.org/document/10714559/.

[4] M. Berlein. dbg. https://github.com/martineberlein/dbg, 2025. Accessed: 2025-09-07.

[5] I. Bouzenia, P. Devanbu, and M. Pradel. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair, Oct. 2024. URL http://arxiv.org/abs/2403.17134. arXiv:2403.17134 [cs].

[6] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching Large Language Models to Self-Debug, Oct. 2023. URL http://arxiv.org/abs/2304.05128. arXiv:2304.05128 [cs].

[7] R. Cheng, M. Tufano, J. Cito, J. Cambronero, P. Rondon, R. Wei, A. Sun, and S. Chandra. Agentic Bug Reproduction for Effective Automated Program Repair at Google, Mar. 2025. URL http://arxiv.org/abs/2502.01821. arXiv:2502.01821 [cs].

[8] A. Contributors. grammars-v4: Grammars written for antlr v4. https://github.com/antlr/grammars-v4, 2025. Accessed: 2025-09-07.

[9] M. Eberlein and K. kwrk1. debugging-benchmark. https://github.com/martineberlein/debugging-benchmark, 2025. Version 0.3.2, MIT License.

[10] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller. Semantic Debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 438–449, San Francisco CA USA, Nov. 2023. ACM. ISBN 979-8-4007-0327-0. doi: 10.1145/3611643.3616296. URL https://dl.acm.org/doi/10.1145/3611643.3616296.

[11] S. Grover, S. Basu, M. Bienkowski, M. Eagle, N. Diana, and J. Stamper. A Framework for Using Hypothesis-Driven Approaches to Support Data-Driven Learning Analytics in Measuring Computational Thinking in Block-Based Programming Environments. *ACM Transactions on Computing Education*, 17(3): 1–25, Sept. 2017. ISSN 1946-6226. doi: 10.1145/3105910. URL https://dl.acm.org/doi/10.1145/3105910.

[12] H. Huynh, Q.-T. Le, T. N. Nguyen, and V. Nguyen. Using LLM for Mining and Testing Constraints in API Testing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 2486–2487, Sacramento CA USA, Oct. 2024. ACM. ISBN 979-8-4007-1248-7. doi: 10.1145/3691620.3695341. URL https://dl.acm.org/doi/10.1145/3691620.3695341.

[13] N. Jiang, X. Li, S. Wang, Q. Zhou, S. B. Hossain, B. Ray, V. Kumar, X. Ma, and A. Deoras. LeDex: Training LLMs to Better Self-Debug and Explain Code, Feb. 2025. URL http://arxiv.org/abs/2405.18649. arXiv:2405.18649 [cs].

[14] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[15] B. A. Jodat, A. Chandar, S. Nejati, and M. Sabetzadeh. Test Generation Strategies for Building Failure Models and Explaining Spurious Failures. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–32, May 2024. ISSN 1049-331X, 1557-7392. doi: 10.1145/3638246. URL https://dl.acm.org/doi/10.1145/3638246.

[16] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1228–1239, New York, NY, USA, Nov. 2020. Association for Computing Machinery. ISBN 978-1-4503-7043-1. doi: 10.1145/3368089.3409687. URL https://dl.acm.org/doi/10.1145/3368089.3409687.

[17] S. Kang, B. Chen, S. Yoo, and J.-G. Lou. Explainable Automated Debugging via Large Language Model-driven Scientific Debugging, Apr. 2023. URL http://arxiv.org/abs/2304.02195. arXiv:2304.02195 [cs].

[18] S. Kang, J. Yoon, and S. Yoo. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction, July 2023. URL http://arxiv.org/abs/2209.11515. arXiv:2209.11515 [cs].

[19] S. Kang, G. An, and S. Yoo. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, July 2024. ISSN 2994-970X. doi: 10.1145/3660771. URL https://dl.acm.org/doi/10.1145/3660771.

[20] T. Le, T. Miller, L. Sonenberg, and R. Singh. Towards the New XAI: A Hypothesis-Driven Approach to Decision Support Using Evidence. In *ECAI 2024*, pages 850–857. IOS Press, 2024. doi: 10.3233/FAIA240571. URL https://ebooks.iospress.nl/doi/10.3233/FAIA240571.

[21] Y. Li, Y. Wu, J. Liu, Z. Jiang, Z. Chen, G. Yu, and M. R. Lyu. COCA: Generative Root Cause Analysis for Distributed Systems with Code Knowledge, Mar. 2025. URL http://arxiv.org/abs/2503.23051. arXiv:2503.23051 [cs].

[22] Z. Liu, C. Chen, J. Wang, M. Chen, B. Wu, X. Che, D. Wang, and Q. Wang. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, Lisbon Portugal, Apr. 2024. ACM. ISBN 979-8-4007-0217-4. doi: 10.1145/3597503.3639180. URL https://dl.acm.org/doi/10.1145/3597503.3639180.

[23] Z. Liu, Y. Ma, J. Xu, J. Ai, X. Gao, H. Sun, and A. Roychoudhury. Agent That Debugs: Dynamic State-Guided Vulnerability Repair, Apr. 2025. URL http://arxiv.org/abs/2504.07634. arXiv:2504.07634 [cs].

[24] Q. Ma, H. Shen, K. Koedinger, and T. Wu. How to Teach Programming in the AI Era? Using LLMs as a Teachable Agent for Debugging. volume 14829, pages 265–279. 2024. doi: 10.1007/978-3-031-64302-6_19. URL http://arxiv.org/abs/2310.05292. arXiv:2310.05292 [cs].

[25] X. Meng, Z. Ma, P. Gao, and C. Peng. An Empirical Study on LLM-based Agents for Automated Bug Fixing, Nov. 2024. URL http://arxiv.org/abs/2411.10213. arXiv:2411.10213 [cs].

[26] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[27] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72, 2025.

[28] A. Ni, M. Allamanis, A. Cohan, Y. Deng, K. Shi, C. Sutton, and P. Yin. NExT: Teaching Large Language Models to Reason about Code Execution, Apr. 2024. URL http://arxiv.org/abs/2404.14662. arXiv:2404.14662 [cs].

[29] P. Rondon, R. Wei, J. Cambronero, J. Cito, A. Sun, S. Sanyam, M. Tufano, and S. Chandra. Evaluating Agent-based Program Repair at Google, Jan. 2025. URL http://arxiv.org/abs/2501.07531. arXiv:2501.07531 [cs].

[30] E. Schluntz and B. Zhang. Building effective agents. https://www.anthropic.com/engineering/building-effective-agents, 2024. Accessed: 2025-08-25.

[31] D. Steinhöfel and A. Zeller. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 583–594, 2022.

[32] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[33] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[34] J. A. Zamudio Amaya, M. Smytzek, and A. Zeller. FANDANGO: Evolving language-based testing. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728915. URL `https://doi.org/10.1145/3728915`.

[35] A. Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann, 2009. URL `https://scholar.google.com/scholar?cluster=7611853054092038992&hl=en&oi=scholarr`.

[36] Z. Zheng, K. Ning, Q. Zhong, J. Chen, W. Chen, L. Guo, W. Wang, and Y. Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(2):50, 2025.

[37] L. Zhong, Z. Wang, and J. Shang. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step-by-step, June 2024. URL `http://arxiv.org/abs/2402.16906`. arXiv:2402.16906 [cs].

Appendix

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 13. September 2025 .....