

SPADE: Synthesizing Data Quality Assertions for Large Language Model Pipelines

Shreya Shankar¹, Haotian Li², Parth Asawa¹, Madelon Hulsebos¹, Yiming Lin¹, J.D. Zamfirescu-Pereira¹, Harrison Chase³, Will Fu-Hinthorn³, Aditya G. Parameswaran¹, Eugene Wu⁴
¹UC Berkeley, ²HKUST, ³LangChain, ⁴Columbia University
 {shreyashankar,pgasawa,madelon,yiminglin,zamfi,adityagp}@berkeley.edu
 haotian.li@connect.ust.hk, {harrison,wfh}@langchain.dev, ewu@cs.columbia.edu

ABSTRACT

Large language models (LLMs) are being increasingly deployed as part of pipelines that repeatedly process or generate data of some sort. However, a common barrier to deployment are the frequent and often unpredictable errors that plague LLMs. Acknowledging the inevitability of these errors, we propose *data quality assertions* to identify when LLMs may be making mistakes. We present SPADE, a method for automatically synthesizing data quality assertions that identify bad LLM outputs. We make the observation that developers often identify data quality issues during prototyping prior to deployment, and attempt to address them by adding instructions to the LLM prompt over time. SPADE therefore analyzes histories of prompt versions over time to create candidate assertion functions and then selects a minimal set that fulfills both coverage and accuracy requirements. In testing across nine different real-world LLM pipelines, SPADE efficiently reduces the number of assertions by 14% and decreases false failures by 21% when compared to simpler baselines. SPADE has been deployed as an offering within LangSmith, LangChain’s LLM pipeline hub, and has been used to generate data quality assertions for over 2000 pipelines across a spectrum of industries.

PVLDB Reference Format:

Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, J.D. Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G. Parameswaran, Eugene Wu. SPADE: Synthesizing Data Quality Assertions for Large Language Model Pipelines. PVLDB, 17(12): 4173 - 4186, 2024. doi:10.14778/3685800.3685835

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/shreyashankar/spade-experiments>.

1 INTRODUCTION

There is a lot of excitement around the use of large language models (LLMs) for processing, understanding, and generating data [16]. Without needing large labeled datasets, one can easily create an *LLM pipeline* for any task on a collection of data items—simply by crafting a natural language *prompt* that instructs the LLM on what to

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
 Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
 doi:10.14778/3685800.3685835

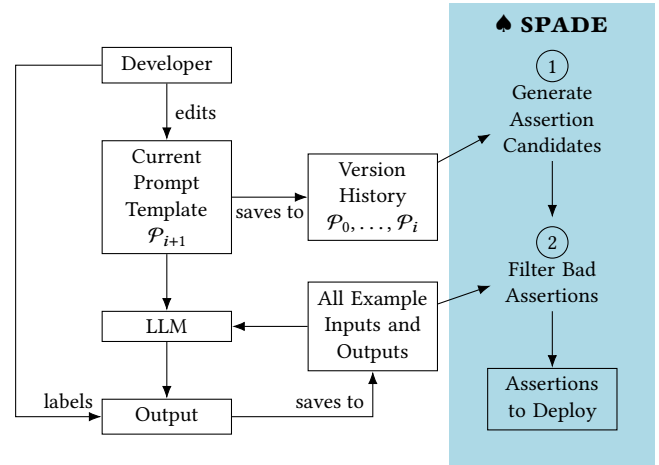


Figure 1: Before a developer deploys a prompt template to production, SPADE analyzes the deltas (i.e., diffs) between consecutive prompt templates to generate assertions. Then, SPADE uses labeled pipeline inputs and outputs to filter out redundant and inaccurate assertions, while maintaining coverage of bad outputs.

do with each item. This could span traditional data processing tasks, such as summarizing each document in a corpus, extracting entities from each news article in a collection of articles, or even imputing missing data for each tuple in a relation [34]. LLMs additionally enable new and more complex data processing tasks that involve generating data, e.g., an LLM could write an explanation for why a product was recommended to a user, author emails to potential sales leads, or draft blog posts for marketing and outreach. In all of these data processing tasks, deploying these LLM pipelines at scale—either offline, on each batch of data items, or online, as and when new items arrive—presents significant challenges, due to data quality errors made by LLMs seemingly at random [23]—with LLMs often disregarding instructions, making mistakes with the output format, or hallucinating facts [48, 63].

One approach to catch errors in deployed LLM pipelines is via *data quality assertions*. Indeed, multiple recent papers [26, 41, 54] and LLM pipeline authoring systems [21, 27, 31] provide mechanisms to embed manually-provided or selected assertions as part of LLM pipelines to catch errors during deployment. However, determining *which* assertions to add remains an open problem—and is a big customer painpoint based on our experience at LangChain—a

company that helps people build LLM pipelines. Developers often find it difficult to determine the right set of assertions for their custom tasks [38]. Challenges include predicting all possible LLM failure modes, the time-consuming nature of writing assertions with various specification methods (like Python functions or LLM calls), the necessity for precision in assertions (especially those involving LLM calls), and the fact that many LLM pipeline developers lack software engineering expertise or coding experience [26, 63]. Moreover, if there are non-informative assertions or too many of them, developers can get overwhelmed monitoring the results of these assertions. While there is some work on automatically detecting errors in traditional ML pipelines [4, 39, 46, 47], this line of work operates on many structured records at a time, and doesn’t apply to an unstructured setting. So, we target the following question: *can we identify data quality assertions for LLM pipelines with as little effort from developers as possible?*

Example LLM Pipeline. Consider an LLM pipeline for a movie streaming platform, where the task is to generate a paragraph of text explaining why a specific movie was recommended to a specific user. A developer might write a prompt template like: “Given the following information about the user, {personal_info}, and information about a movie, {movie_info}: write a personalized note for why the user should watch this movie” to be executed for many user-movie pairs. In theory, this prompt seems adequate, but the developer might observe some data quality issues while testing it across different inputs: the LLM output might reference a movie the user never watched, cite a sensitive attribute (e.g., race or ethnicity), or even exhibit a basic issue by being too short. To fix these problems, developers typically add instructions to the prompt to catch these data quality issues, such as “Don’t reference race in your answer”. However, the LLM may still violate these instructions in an unpredictable manner for some data items, requiring assertions applied to LLM outputs post-hoc, during deployment.

Analyzing Prompt Version Histories. To automatically synthesize assertions for developers, our first insight is that we can mine prompt version histories to identify assertion criteria for LLM pipelines, since developers implicitly embed data quality requirements through changes to the prompt, or *prompt deltas*, over time. In our example above, instructions such as “Make sure your response is at least 3 sentences” or “Don’t reference race in your answer” could each correspond to a candidate assertion. To understand the types of prompt deltas in LLM pipelines and verify their usefulness for data quality assertions, we present an analysis of prompt version histories of 19 custom pipelines from LangChain users. Using this analysis, we construct a taxonomy of prompt deltas (Figure 2), which may be of independent interest for researchers and practitioners studying how to best build LLM pipelines.

Redundancy in Data Quality Assertions. Our second insight from analyzing these pipelines is that if we were to create candidate assertions from prompt deltas, say, using an LLM, there may be far too many assertions—often exceeding 50 for just a few prompt deltas. Many of these assertions (or equivalently, prompt deltas) are redundant, while some are too imprecise or ambiguous to be useful (e.g., “return a concise response”). Reducing this redundancy is nontrivial, even for engineers: assertions themselves can involve

LLM calls with varying accuracies, motivating an automated component that can filter out bad candidates. One approach is to use a small handful of developer-labeled LLM outputs to estimate each data quality assertion’s false positive and false negative rate—but collectively determining the right *set* of assertions that can catch most errors, while incorrectly flagging as few outputs as possible, is not straightforward.

SPADE: Our Data Quality Assertion Generation Framework. In this paper, we leverage these insights to develop SPADE (System for Prompt Analysis and Delta-Based Evaluation, Figure 1)¹. SPADE’s goal is to select a set of boolean assertions with minimal overlap, while maximizing coverage of bad outputs and minimizing false failures (correct outputs that are incorrectly flagged) for the conjunction of selected assertions. We decompose SPADE into two components: candidate assertion generation and filtering.

Component 1: Prompt Deltas for Candidate Assertion Generation. For generating candidate assertions, instead of directly querying an LLM to “write assertions for x prompt,” which causes the LLM to miss certain portions of the prompt, we generate candidates from each prompt delta, which typically indicate specific failure modes of LLMs. SPADE leverages the aforementioned taxonomy of prompt deltas we constructed by first automatically categorizing deltas using the taxonomy, then synthesizing Python functions (that may include LLM calls) as candidate assertions. ***With LangChain, we publicly release this component of SPADE—which has been subsequently used for over 2000 pipelines across more than 10 sectors like finance, medicine, and IT [51].*** We present an analysis of this usage in Section 2.4.

Component 2: Filtering Candidate Assertions with Limited Data. To filter out incorrect and redundant candidate assertions, instead of requiring cumbersome manual selection or even fine-tuning of separate models [43, 57], we propose an automated approach that only requires a small handful of labeled examples, which are usually already present in most target applications. Using these examples, we could estimate each assertion’s false failure rate (FFR), i.e., how often an assertion incorrectly flags failures, and eliminate individual assertions that exceed a given threshold. However, given we are selecting a set of assertions, the set may still exceed the FFR threshold and flag too many failures incorrectly, and redundancies may persist. We show that selecting a small subset of assertions to meet failure coverage and FFR criteria is NP-hard. That said, we may express the problem as an integer linear program (ILP) and use an ILP solver to identify a solution in a reasonable time given the size of our problem (hundreds to thousands of variables).

In some cases, when there are limited developer-provided examples, we find that labeled LLM outputs may not cover all failure modes, leading to omission of valuable data quality assertions. For instance, in our movie recommendation scenario, an assertion that correctly verifies if the output is under 200 words will get discarded if all outputs in our developer-labeled sample respect this limit. To expand coverage, active learning and weak supervision approaches can be used to sample and label new LLM input-output pairs for each candidate assertion [6, 40], but this may be expensive or inaccessible for non-programmers. We introduce assertion *subsumption* as a way of ensuring comprehensive coverage: if one

¹We called our framework SPADE since it helps users dig up or unearth useful assertions.

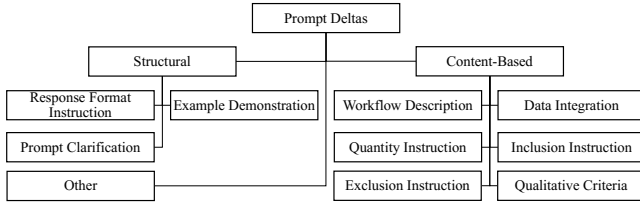


Figure 2: Taxonomy of prompt deltas from 19 LLM pipelines.

assertion doesn’t encompass the failure modes of another, both may be selected. As such, SPADE selects a minimal set of assertions that respects failure coverage, accuracy, *and* subsumption constraints.

Overall, we make the following contributions:

- We identify prompt version history as a rich source for LLM output correctness, creating a taxonomy of assertion criteria from 19 diverse LLM pipelines (Section 2),
- We introduce SPADE, our system that automatically generates data quality assertions for LLM pipelines. In a public release of SPADE’s candidate assertion generation component², we observe and analyze 2000+ deployments (Section 2.4),
- We present a method to select a minimal set of assertions while meeting coverage and accuracy requirements, used by SPADE to reduce the number of assertions. For low-data settings, we introduce assertion subsumption as a novel proxy for coverage (Section 3), and
- We demonstrate SPADE’s effectiveness on nine real-world LLM pipelines (eight of which we open-source). In our low-data setting (approximately 75 inputs and outputs per pipeline), our subsumption-based solution outperforms simpler baselines that do not consider interactions between assertions by reducing the number of assertions by 14% and lowering the false failure rate by 21% (Section 4).

2 IDENTIFYING CANDIDATE ASSERTIONS

Our first goal is to generate a set of candidate assertions. We describe how *prompt deltas* can inform candidate assertions and explain how to derive candidate assertions from them.

2.1 Prompt Deltas

A single-step LLM pipeline consists of a prompt template \mathcal{P} , which is formatted with a serialized input tuple t to derive a prompt that is fed to an LLM, returning a response. There can be many versions of \mathcal{P} , depending on how a developer iterates on their prompt template. Let \mathcal{P}_0 be the empty string, the 0th version, and let \mathcal{P}_i be the i th version of a template. In the movie recommendation example from the introduction, suppose there are 7 versions, where \mathcal{P}_7 is the following: “Given the following information about the user, {personal_info}, and information about a movie, {movie_info}: write a personalized note for why the user should watch this movie. Ensure the recommendation note is concise, not exceeding 100 words. Mention the movie’s genre and any shared cast members between the {movie_name} and other movies the user has watched. Mention any awards or

critical acclaim received by {movie_name}. Do not mention anything related to the user’s race, ethnicity, or any other sensitive attributes.”

We define a prompt delta $\Delta\mathcal{P}_{i+1}$ to be the *diff* (or difference) between \mathcal{P}_i and \mathcal{P}_{i+1} . Concretely, a prompt delta $\Delta\mathcal{P}$ is a set of sentences, where each sentence is tagged as an addition (i.e., “+”) or deletion (i.e., “-”). Table 1 shows the $\Delta\mathcal{P}$ s for a number of versions for our example. Each sentence in $\Delta\mathcal{P}_i$ is composed of additions (i.e., new sentences in \mathcal{P}_i that didn’t exist in \mathcal{P}_{i-1}) and deletions (i.e., sentences in \mathcal{P}_{i-1} that don’t exist in \mathcal{P}_i). A modification to a sentence is represented by a deletion and addition. Each addition in $\Delta\mathcal{P}_i$ indicates possible assertion criteria, as shown in the right-most column of Table 1.

2.2 Prompt Delta Analysis

To understand what assertions developers may care about, we turn to real-world LLM pipelines. We analyzed 19 LLM pipelines collected from LangChain users, each of which consists of between three and 11 historical prompt template versions. These pipelines span various tasks across more than five domains (e.g., finance, marketing, coding, education, health), from generating workout summaries to a chatbot acting as a statistics tutor. Details of the pipelines can be found in our tech report [50]. For each pipeline, we categorized prompt deltas, i.e., $\Delta\mathcal{P}_i$, into different types—for example, instructing the LLM to include a new phrase in each response (i.e., inclusion), or instructing the LLM to respond with a certain tone (i.e., qualitative criteria). Two authors iterated on the categories 4 times through a process of open and axial coding, ultimately producing the taxonomy in Figure 2. The taxonomy-annotated dataset of prompt versions can be found online³.

We divide deltas into two main high-level categories: Structural and Content-Based. Around 35% categories identified across all deltas in our dataset were Structural, and 65% were Content-Based. Structural deltas indicate a minor restructuring of the prompt, without changing any criteria of a good response (e.g., adding a newline for readability), or specification of the intended output (e.g., JSON or Markdown). Plausible assertion criteria based on structural deltas would check if the LLM output adheres to the user-specified structure. On the other hand, content-based deltas indicate a change in the *meaning* or definition of the task. Content-based deltas include descriptions of the workflow steps that the LLM should perform (e.g., “first, do X, then, come up with Y”), instructions of specific phrases to include or exclude in responses, or qualitative indicators of good responses (e.g., “maintain a professional tone”). The Data Integration subcategory (under Content-Based deltas) concerns adding new sources of context to the prompt—for example, adding a new variable like “{movie_info}” to the prompt, indicating a new type of information to be analyzed along with other content in the prompt. For some illustrative examples of prompt deltas for each category, we categorize the prompt deltas in Table 1, and in Table 2, we show sample prompt deltas for each category in our taxonomy. This taxonomy may be of independent interest to researchers studying the process of prompt engineering, as well as practitioners seeking to identify ways to improve their prompts for production LLM pipelines.

²<https://spade-beta.streamlit.app>

³https://github.com/shreyashankar/spade-experiments/blob/main/taxonomy_labels.csv

Table 1: Comparison of 7 prompt versions for an LLM pipeline to write personalized movie recommendations. In each $\Delta\mathcal{P}_i$, a sentence starts with “+” if it is a newly added sentence; a sentence starts with “-” if it is removed from \mathcal{P}_{i-1} . For each $\Delta\mathcal{P}_i$, we list its category (using the taxonomy in Figure 2) and possible new assertion criteria.

Version i	$\Delta\mathcal{P}_i$	$\Delta\mathcal{P}_i$ Category	Possible New Assertion Criteria
1	+ Given the following information about the user, {personal_info}, and information about a movie, {movie_info}: write a personalized note for why the user should watch this movie.	Inclusion	Response should be personalized and relevant to the given user information
2	+ Include elements from the movie’s genre, cast, and themes that align with the user’s interests.	Inclusion	Response includes specific references to the user’s interests related to the movie’s genre, cast, and themes
3	+ Ensure the recommendation note is concise.	Qualitative Assessment	Response should be concise
4	- Ensure the recommendation note is concise. + Ensure the recommendation note is concise, not exceeding 100 words.	Count	Response should be within the 100 word limit
5	- Include elements from the movie’s genre, cast, and themes that align with the user’s interests. + Mention the movie’s genre and any shared cast members between the {movie_name} and other movies the user has watched.	Inclusion	Response should mention genre and verify cast members are accurate
6	+ Mention any awards or critical acclaim received by movie_name.	Inclusion	Response should include references to awards or critical acclaim of the movie
7	+ Do not mention anything related to the user’s race, ethnicity, or any other sensitive attributes.	Exclusion	Response should not include references to sensitive personal attributes

Overall, our exercise in building this taxonomy reveals two key findings. First, developers across diverse LLM pipelines iterate on prompts in similar ways as encoded as nodes in the taxonomy, many of which correspond to aspects that can be explicitly checked via assertions. Second, we find that an automated approach to synthesize data quality assertions may be promising, since our taxonomy reveals several such instances where the prompt delta could directly correspond to an assertion that captures the same requirement. For example, many deltas correspond to instructions to include or exclude specific phrases, indicating that developers may benefit from assertions that explicitly verify the inclusion or exclusion of such phrases in LLM outputs.

However, if we are to use an LLM to automatically generate assertion criteria based on our taxonomy of deltas, we also need to test whether LLMs can accurately identify the categories corresponding to a delta. Therefore, we confirmed GPT-4’s correct categorization of prompt deltas (as of October 2023): we assigned ground truth categories to all prompt deltas from the 19 pipelines, and GPT-4 achieved an F1 score of 0.8135. The prompt used for category extraction from prompt deltas is detailed in our tech report [50].

2.3 From Taxonomy to Assertions

As we saw previously, prompt delta as categorized into nodes in our taxonomy often correspond to meaningful assertion criteria. Next, we need a method to automatically synthesize the data quality assertions from the prompt deltas. A natural idea is to prompt an LLM to generate assertion functions corresponding to relevant categories in our taxonomy given the prompt deltas. We tried this approach with GPT-4 in January 2024 and observed several omitted assertions that clearly corresponded to categories in our taxonomy. Therefore, we adopt a two-step prompting process in our approach, as it has been demonstrated that breaking tasks into steps can enhance LLM accuracy [20, 59, 61]. In the first step, we prompt

Table 2: Categories of prompt deltas.

Category	Explanation	Example Prompt Delta
Response Format Instruction	Structure guidelines.	“+ Start response with ‘You might like...’”
Example Demonstration	Illustrative example.	“+ For example, here is a response for sci-fi fans...”
Prompt Clarification	Refines prompt/removes ambiguity.	“- Discuss/+ Explain movie fit...”
Workflow Description	Describe “thinking” process.	“+ First, analyze viewing history...”
Data Integration	Adds placeholders.	“+ Include user’s {genre} reviews.”
Quantity Instruction	Adds numerical content.	“+ Keep note under 100 words.”
Inclusion Instruction	Directs specific content.	“+ Mention movie awards/acclaim.”
Exclusion Instruction	Advises on omissions.	“+ Avoid movie plot spoilers.”
Qualitative Criteria	Sets stylistic attributes.	“+ Maintain friendly, positive tone.”

GPT-4 for natural language descriptions of assertion criteria, and in the second step, we prompt GPT-4 to generate Python functions that implement such criteria.

More specifically, for each prompt delta $\Delta\mathcal{P}_i$, we first prompt an LLM to suggest as many criteria as possible for assertions—each aligning with a taxonomy category from Figure 3. A criterion is loosely defined as some natural language expression that operates on a given output or example and evaluates to True or False (e.g., “check for conciseness”). Our method analyzes every $\Delta\mathcal{P}_i$ instead of just the last prompt version for several reasons: developers often remove instructions from prompts to reduce costs while expecting the same behavior [38], prompts contain inherent ambiguities and imply multiple ways of evaluating some criteria, and complex prompts may lead to missed assertions if only one version is analyzed. Consequently, analyzing each $\Delta\mathcal{P}_i$ increases the likelihood of generating relevant assertions.

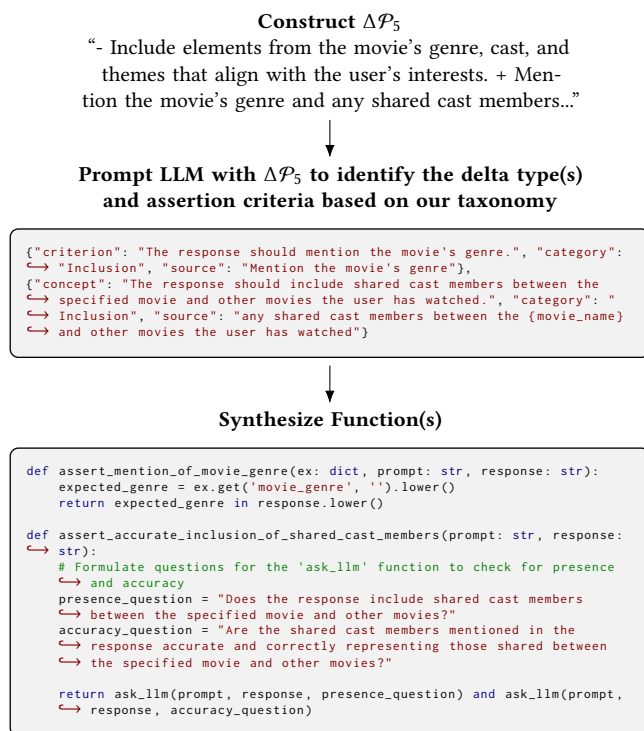


Figure 3: Generating candidate assertions from a $\Delta\mathcal{P}$.

For each delta, our method collects the criteria identified and prompts an LLM again to create Python assertion functions. The synthesized functions can use external Python libraries or pose binary queries to an LLM for complex criteria. For function synthesis, the LLM is instructed that if the criterion is vaguely specified or open to interpretation, such as “check for conciseness,” it can generate multiple functions that each evaluate the criterion. In this conciseness example, the LLM could return multiple functions—a function that splits the response into sentences and ensures that there are no more than, say, 3 sentences, a function that splits the response into words and ensures that there are no more than, say, 25 words, or a function that sends the response to an LLM and asks whether the response is concise. The overall outcome of this is a multiset of candidate functions $F = \{f_1, \dots, f_m\}$. The two prompts for generating assertion criteria and functions can be found in our tech report [50].

2.4 Initial Deployment

To assess the potential of our auto-generated assertions, in November 2023, we released an early prototype of SPADE’s candidate assertion generation framework via a Streamlit application⁴. In the Streamlit app, a developer can either paste their prompt template that they want to generate assertions for, or they can point to their LangChain Hub prompt template (which contains prompt version

histories via commits). The app then visualizes the identified taxonomy categories in the user’s prompt and displays the candidate assertions, as shown in the screenshot in Figure 4.

Tool Usage Insights and Feedback. From the reception to our Streamlit application, we found significant interest in auto-generated data quality assertions for LLM pipelines: there have been over 2000 runs of the app for custom prompt templates (i.e., not the sample default prompt template in app). These runs span many fields, including medicine, education, cooking, and finance—providing insights from a diversity of use cases for LLM pipelines. Figure 5 shows a rough breakdown of the use cases people wanted to generate assertions for; however, it’s important to note that some runs may not cleanly fit into a single category (e.g., a chatbot for telehealth-related questions for a medical provider could be in “customer support” and “health”). Interestingly, 8% of tasks related to conversational assistants, and we observed instances of at least four different companies generating assertions for their chatbots, given that the company name was in the prompt. Users for 48 of these runs clicked the “download assertions” button, which downloads the candidate assertions as a Python file. We note that users can also directly copy the assertions code displayed, instead of downloading the assertions as a Python file, and we did not measure the copy events. No users clicked the “thumbs down” button—which is not to say that the generated assertions were perfect. After our Streamlit app was released, we found an unprompted review of SPADE-generated assertions from a LangChain user who built a “chat-with-your-pdf” tool⁵: in their words, “When I saw it I didn’t believe it could work that well, but it really did and made the evaluation process fun and ez.” We also found an independently-written open-source re-implementation of SPADE⁶.

Observations about Assertion Criteria. Across the 2000+ runs, with regards to our taxonomy, assertion criteria were most commonly derived from inclusion and exclusion instructions. For example, for a shopping assistant, where the objective was to find the most relevant product related to a customer’s query, responses were required to include “all the features [the customer] asked for.” In another customer support agent example, responses were required to include “exact quotes in the [context] relevant to the [customer’s] question...word for word.” One common exclusion instruction across chat-related tasks was to avoid any discussion unrelated to the end-user’s question; however, we noticed that GPT-4 struggled to generate assertion criteria around such a generic exclusion instruction. We found Python functions generated to implement criteria such as “avoid unrelated ideas”, did not use an LLM to check the criteria. Instead, they checked for the presence of specific phrases like “unrelated” and “I’m sorry, I did not find anything related” in the response. Such errors indicate the limitations of current state-of-the-art LLMs and may suggest the need for specialized, fine-tuned LLMs for generating assertions in future work; here, we work with the limitations of present-day state-of-the-art LLMs. We discuss this further in our tech report [50].

A new pattern we noticed after deployment, which wasn’t seen in our initial analysis, is that developers often wish to hide certain

⁴<https://spade-beta.streamlit.app/>

⁵https://twitter.com/th_calafatidis/status/1728144652119769394

⁶https://github.com/uptrain-ai/uptrain/blob/main/examples/integrations/spade/evaluating_guidelines_generated_by_spade.ipynb

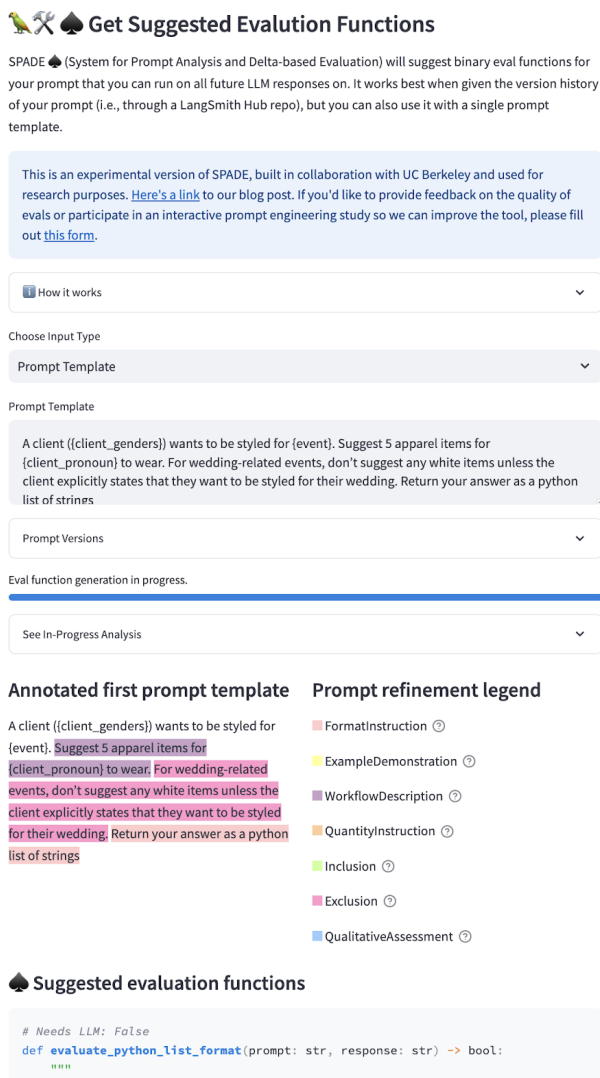


Figure 4: Screenshot of an early version of SPADE.

parts of the LLM workflow in the outputs. This could be viewed as a special instance of the exclusion category in our taxonomy. For instance, in a prompt related to enterprise automation, the first instruction was to write a query targeting a specific database (e.g., “Write a SQL query to fetch the most relevant table from the MySQL database”). However, another instruction in the prompt specified that the name of the database should not appear in the final summary returned to the end-user (e.g., “Do not mention that you queried the MySQL table X”). We were pleasantly surprised to find that our process for generating criteria successfully identified and implemented such exclusion instructions accurately.

Observations about Assertions. In LLM pipelines with numerous prompt versions, we observed two main patterns. First, prompt engineering often leads to many similar assertions, and redundancy can be a headache at deployment when a developer has to keep track of so many assertions. For instance, a pipeline to summarize lecture

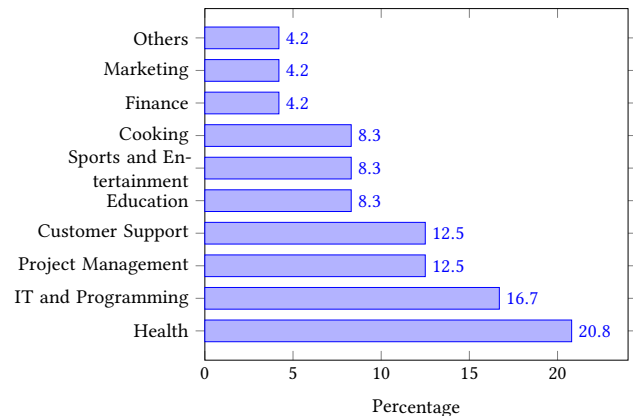


Figure 5: Prompts submitted to the SPADE Streamlit application span a variety of fields.

transcripts⁷ had 14 prompt versions, with many edits localized to the paragraph providing instructions on titles, speakers, and dates, creating multiple overlapping assertions. 10 assertions assessed the lecture’s central thesis; two are as follows:

```

async def assert_response_articulates_central_thesis(
    example: dict, prompt: str, response: str
):
    return "Central Thesis:" in response

async def assert_response_completeness(example: dict, prompt: str, response:
    str):
    required_elements = [
        "Context:",
        "Central Thesis:",
        "Key Points:",
        "Conclusions and Takeaways:",
        "Glossary of Important Terms:",
    ]
    return all(element in response for element in required_elements)

```

Second, many assertions may be incorrect, causing runtime errors or false failures. Assessing the accuracy of these assertions is challenging, particularly for those that are complex or invoke LLMs themselves, where even experienced developers might not be able to gauge their effectiveness without viewing the results of the functions on many examples. Even if the assertions are not incorrect, they still may have undesirable failure rates or coverage, which is often hard for a developer to reason about in conjunction with other assertions. Since there can be 50+ assertions generated for just a handful of prompt versions (as demonstrated in Section 4), manually filtering them by eyeballing failure rates for each subset of assertions is impractical. Therefore, we adopt an automated approach for filtering, as discussed in the following section.

3 FILTERING CANDIDATE ASSERTIONS

As we saw in the previous section, we often have redundant and incorrect assertions, particularly in pipelines with numerous prompt versions. Here, we focus on filtering this candidate set to a smaller number, which not only improves efficiency and reduces cost when deploying the assertions to run in production, but it also reduces cognitive overhead for the developer.

⁷<https://smith.langchain.com/hub/kirby/simple-lecture-summary>

3.1 Definitions

Consider e_i as an example end-to-end execution (i.e., run) of an LLM pipeline on some input. For the purposes of this section, we assume the prompt template \mathcal{P} is fixed to be the final prompt version that the developer eventually decided on. We denote $y_i \in \{0, 1\}$ to represent whether the developer considers e_i to be a *success* (1) or *failure* (0).

Let E be the set of all such example runs (this set is not provided upfront, as we will deal with shortly). We define an assertion function $f : E \rightarrow \{0, 1\}$, where 1 indicates that the example was processed successfully by the LLM and 0 otherwise. Let $F' = \{f_1, f_2, \dots, f_k\}$ be a set of k data quality assertions. An example e_i is deemed *successful* by F' if and only if it satisfies all assertions in F' . Specifically:

$$\hat{y}_i = \begin{cases} 1 & \text{if } f(e_i) = 1 \quad \forall f \in F', \quad // \text{Deemed successful by all of } F' \\ 0 & \text{otherwise.} \quad // \text{Deemed a failure by } \geq 1 \text{ of } F' \end{cases}$$

Given all m candidate assertions $F = \{f_1, f_2, \dots, f_m\}$, the objective is to select $F' \subseteq F$ such that $\hat{y}_i = y_i$ for most examples in E , with F' being as small as possible. This goal involves maximizing failure coverage and minimizing the false failure rate and selected function count, expressed as follows:

Definition 3.1. Coverage for a set F' is the proportion of actual failures that are correctly identified by F' , defined as:

$$\text{Coverage}(F') = \frac{\sum_i \mathbb{I}[\hat{y}_i = 0 \wedge y_i = 0]}{\sum_i \mathbb{I}[y_i = 0]} \quad // \text{Failures caught by } F'$$

Definition 3.2. False Failure Rate (FFR) for a set F' is the fraction of examples that F' incorrectly evaluates as failures ($\hat{y}_i = 0$) when they are actually successful ($y_i = 1$), defined as:

$$\text{FFR}(F') = \frac{\sum_i \mathbb{I}[\hat{y}_i = 0 \wedge y_i = 1]}{\sum_i \mathbb{I}[y_i = 1]} \quad // \text{Non-failures flagged by } F'$$

In both definitions above, \hat{y}_i represents set F' 's prediction for the i -th example, y_i is the actual outcome, while \mathbb{I} is the indicator function. In practice, coverage and FFR are impossible to compute since the universe of examples E is unknown. So, for now, we assume access to a subset $E' \subset E$ of labeled LLM responses, where E' is a manually provided set of example runs and may not contain all the types of failures the LLM pipeline could observe—an issue we will deal with in Section 3.3. Thus, we replace Definition 3.1 and Definition 3.2 with $\text{Coverage}_{E'}(F')$ and $\text{FFR}_{E'}(F')$, omitting the subscript E' for brevity.

3.2 Coverage Problem Formulation

Our goal is therefore to select a minimal set of assertions $F' \subseteq F$ based on a sample $E' = \{e_1, \dots, e_n\} \subset E$. Formally:

minimize $|F'|$

subject to: $\text{Coverage}(F') \geq \alpha, \quad \text{FFR}(F') \leq \tau$

The above problem may be infeasible with certain values of α , for multiple reasons. To see this, consider the case where no assertion catches a specific failure, while α is set to 1. We ignore this (and similar cases) for now and defer their discussion to Section 4. To expand out the above, we introduce a matrix M (size $n \times m$) to track each assertion's result on each example e_i , where $M_{ij} = 1$

if $f_j(e_i) = 1$ (i.e., F_j deems e_i a success) and $M_{ij} = 0$ otherwise. We also define binary variables x_j and w_{ij} to represent whether an assertion is chosen and if it marks an example as a failure: $w_{ij} = (1 - M_{ij}) \cdot x_j$, which is based on whether f_j denotes e_i as a failure and f_j is included in F' . We additionally introduce the binary variable u_i to represent whether a failed example is covered by any selected assertion:

$$u_i \leq \sum_{j=1}^m w_{ij}, \quad \forall i \in [1, n] : y_i = 0. \quad // \text{Coverage of failure } e_i$$

Then, the coverage constraint can be written as:

$$\frac{\sum_{i: y_i=0} u_i}{\sum_i \mathbb{I}[y_i = 0]} \geq \alpha.$$

FFR is decomposed similarly: we introduce a new binary variable z_i , which defines whether F' denotes e_i as a failure while e_i is actually a successful example (i.e., false failure):

$$z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n] ; \forall j \in [1, m] \quad // e_i \text{ is a false failure}$$

Then, the FFR constraint is:

$$\frac{\sum_{i=1}^n z_i}{\sum_{i=1}^n \mathbb{I}[y_i = 1]} \leq \tau.$$

We can then state the problem of minimizing the number of assertions while meeting E' coverage and FFR constraints as an Integer Linear Program (ILP):

$$\text{minimize} \quad \sum_{j=1}^m x_j$$

subject to: $w_{ij} = (1 - M_{ij}) \cdot x_j, \quad \forall i \in [1, n], \forall j \in [1, m];$

$$u_i \leq \sum_{j=1}^m w_{ij}, \quad \forall i \in [1, n] \text{ where } y_i = 0; \quad \frac{\sum_{i: y_i=0} u_i}{\sum_i \mathbb{I}[y_i = 0]} \geq \alpha;$$

$$z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n], \forall j \in [1, m]; \quad \frac{\sum_{i=1}^n z_i}{\sum_{i=1}^n \mathbb{I}[y_i = 1]} \leq \tau;$$

$$x_j, u_i, z_i, w_{ij} \in \{0, 1\}, \quad \forall i \in [1, n], \forall j \in [1, m].$$

We refer to a solution for this ILP as SPADECov . Trivially, the problem is NP-hard for $\tau = 0$ and $\alpha = 1$, via a simple reduction from set cover, and is in NP, since it can be stated in ILP form. In our case, given tens of candidate assertions and fewer than 100 examples e_i , the ILPs tend to be of reasonable size (i.e., thousands of variables). Most ILP solvers can efficiently and quickly such programs.

3.3 Subsumption Problem Formulation

So far, we've assumed that the developer is willing to provide a comprehensive set of labeled example runs E' . In settings where the developer is unwilling to do so, and where E' does not include all failure types in E , SPADECov may overlook useful assertions in F that only catch failures in $E \setminus E'$ —as shown empirically in Section 4. We initially considered using active learning [6] to sample more LLM responses for each assertion and weak supervision to label the responses [40]. However, this approach can be costly with state-of-the-art LLMs, and it demands significant manual effort to balance failing and successful examples for each assertion, ensuring meaningful FFRs, and avoiding the exclusion of assertions due to underrepresented failure types.

For this setting, we additionally introduce the notion of *subsumption*. To illustrate this concept, consider the following real example of assertions from a pipeline which uses an LLM to review pull requests to Github codebases: `gratitude_v1` checks for the presence of “thank you” in the response; `gratitude_v2` verifies if the response contains “thank you” or “thanks”; and `gratitude_v3` checks for “thank you”, “thanks”, or “grateful” in the response. In this hierarchy, `gratitude_v3` subsumes the other two assertions, as it covers all their cases and more.

Assuming that all candidate assertion functions cover as many failure modes as possible, our goal is to pick $F' \subseteq F$ such that assertions in $F \setminus F'$ are *subsumed* by F' . Formally, a set of functions S subsumes some function f if the conjunction of functions in S logically implies the conjunction of functions in S and f . That is,

Definition 3.3. A set of functions $S \implies f$ if and only if $\forall e \in E, \exists s \in S$ such that $s(e) \implies f(e)$. In other words, if $S \implies f$, then f catches no new failures that S does not already catch.

A simple example of subsumption is as follows: suppose our set of functions S contains only one function f , which parses an LLM output into a JSON list to check that the list has at least 2 elements. f may be generated as a result of a count-related instruction from our taxonomy of prompt deltas. Now, let g be some other function that only checks if the output can be turned into a JSON list. g might have been generated due to a “Response Format Instruction”-type delta. If g fails for some LLM output, f —and therefore S —must also fail that output; thus $S \implies g$.

3.3.1 ILP with Subsumption Constraints. We reformulate the problem with subsumption constraints. Let G be the set of functions in $F \setminus F'$ not subsumed by F' :

$$\text{minimize } |F'| + |G|$$

$$\text{subject to: } \text{Coverage}(F') \geq \alpha, \quad \text{FFR}(F') \leq \tau$$

To represent G , we introduce binary variables and a matrix K to denote subsumption relationships. $K_{ij} = 1$ if and only if $f_i \implies f_j$. We discuss how we construct K in more detail in Section 3.3.2. Recall that x_j represents whether f_j is selected in F' . For each function f_j , a binary variable r_j indicates if it is subsumed by F' .

$$x_i \cdot K_{ij} \leq r_j \leq \sum_{\substack{i=1 \\ i \neq j}}^m (x_i \cdot K_{ij}), \quad \forall j \in [1, m], i \neq j.$$

Now, s_j will denote if f_j is neither in F' nor subsumed by F' . We break this into three parts. First,

$$s_j \leq 1 - x_j, \quad \forall j \in [1, m],$$

indicates that $f_j \notin F'$ if s_j is allowed to be 1 (i.e., $f_j \notin F'$ when $x_j = 0$). The second constraint,

$$s_j \leq 1 - r_j, \quad \forall j \in [1, m],$$

captures the condition where f_j is not subsumed by F' . Here, if $r_j = 0$, suggesting no subsumption, then s_j may be 1. Lastly, to combine these conditions, we employ the constraint,

$$s_j \geq 1 - x_j - r_j, \quad \forall j \in [1, m],$$

which ensures that s_j can only be 1 if both prior conditions are satisfied: f_j is neither in F' nor subsumed by F' .

Finally, our objective is to minimize the sum of the number of functions in F' and non-subsumed functions G . The ILP formulation then becomes (with changes highlighted in blue):

$$\text{minimize } \sum_{j=1}^m x_j + \sum_{j=1}^m s_j$$

$$\text{subject to: } w_{ij} = (1 - M_{ij}) \cdot x_j, \quad \forall i \in [1, n], \forall j \in [1, m];$$

$$u_i \leq \sum_{j=1}^m w_{ij}, \quad \forall i \in [1, n] \text{ where } y_i = 0; \quad \frac{\sum_{i: y_i=0} u_i}{\sum_i \mathbb{I}[y_i = 0]} \geq \alpha;$$

$$z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n], \forall j \in [1, m]; \quad \frac{\sum_{i=1}^n z_i}{\sum_{i=1}^n \mathbb{I}[y_i = 1]} \leq \tau;$$

$$x_i \cdot K_{ij} \leq r_j \leq \sum_{\substack{i=1 \\ i \neq j}}^m (x_i \cdot K_{ij}), \quad \forall j \in [1, m], i \neq j;$$

$$s_j \leq 1 - x_j, \quad s_j \leq 1 - r_j, \quad \forall j \in [1, m];$$

$$s_j \geq 1 - x_j - r_j, \quad \forall j \in [1, m];$$

$$x_j, u_i, z_i, w_{ij}, r_j, s_j \in \{0, 1\}, \quad \forall i \in [1, n], \forall j \in [1, m].$$

We call a solution to this ILP $\text{SPADE}_{\text{sub}}$. We maintain the coverage constraint because the subsumption approach alone does not inherently account for the distribution or the significance of different types of failures. For instance, if a particular type of failure makes up a critical portion of E' , a subsumption-based approach might overlook it. To see this in the simplest case, consider $\alpha = 1$: simply optimizing for the sum $|F'| + |G|$ does not guarantee all failures in E' are covered. In practice, $\text{SPADE}_{\text{sub}}$ is less sensitive to α than $\text{SPADE}_{\text{cov}}$, as we will discuss further in Section 4.

3.3.2 Assessing Subsumption. Here, we detail how to construct K , our matrix representing subsumption relationships between pairs of functions f_i, f_j . For pure Python functions, one could use static analysis to determine subsumption. However, it becomes complex when dealing with assertions that include LLM calls or a mix of pure Python and LLM-invoking assertions. For these, SPADE employs GPT-4 to identify potential subsumptions $\{a\} \implies b$ for pairs of functions a, b . For any pipeline, there are only two LLM calls to determine all subsumptions: first, all assertion functions are combined into a single prompt for GPT-4, instructing it to list as many subsumption relationships as possible, then prompting again to transform its response into a parse-able list of pairs $a \implies b$.

To maximize precision of \implies relationships identified, we employ some heuristics. First, E' can filter subsumptions: for f_i and f_j , observe that:

$$\exists e_i \in E' : (f_i(e_i) = 1) \wedge (f_j(e_i) = 0) \Rightarrow (\{\dots, f_i, \dots\} \not\Rightarrow f_j).$$

In other words, any set containing f_i definitely does not subsume f_j if f_j flags a failure that $\{f_i\}$ does not. Next, we use the FFR threshold to skip evaluating subsumption. Observe that, for any set of assertions S and $f \notin S$,

$$\begin{aligned} \max(\text{FFR}(S), \text{FFR}(\{f\})) &\leq \text{FFR}(S \cup \{f\}), \\ \text{FFR}(S \cup \{f\}) &\leq \text{FFR}(S) + \text{FFR}(\{f\}). \end{aligned} \quad (1)$$

As such, we need not evaluate $\{f_i\} \implies f_j$ if either $\text{FFR}(\{f_i\}) \geq \tau$ or $\text{FFR}(\{f_j\}) \geq \tau$. Lastly, we use transitivity of implication to

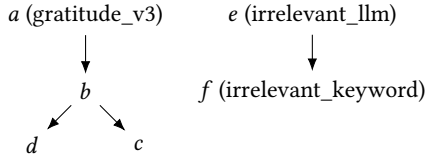


Figure 6: Example depicting subsumption relationships. Node a represents the most comprehensive assertion to check for presence of gratitude, e.g., “thank you”, in the LLM response (gratitude_v3 in the beginning of Section 3.3), subsuming nodes b , c , and d , which are less comprehensive checks (e.g., checking fewer keywords). Node e represents an LLM-based check for irrelevant content in the LLM response, subsuming node f , a simpler keyword-based check that is less comprehensive.

further prune checks: if $x \implies y$ and $y \implies z$, then $x \implies z$ is also true.

3.3.3 Subsumption without Examples. For completeness, we also consider the case where we have no developer-provided examples E' . In this case, we are only reliant on our subsumption relationships to pick our set F' . Our problem can then be restated as follows:

$$\begin{aligned}
 &\text{minimize} \quad \sum_{j=1}^m x_j + \sum_{j=1}^m s_j \\
 &\text{subject to:} \quad x_i \cdot K_{ij} \leq r_j \leq \sum_{\substack{i=1 \\ i \neq j}}^m (x_i \cdot K_{ij}), \quad \forall j \in [1, m]; \\
 &\quad s_j \leq 1 - x_j, \quad s_j \leq 1 - r_j, \quad \forall j \in [1, m]; \\
 &\quad s_j \geq 1 - x_j - r_j, \quad \forall j \in [1, m]; \\
 &\quad x_j, r_j, s_j \in \{0, 1\}, \quad \forall j \in [1, m].
 \end{aligned}$$

However, it turns out this problem is no longer NP-Hard. Consider the example in Figure 6, where an edge indicates subsumption. For example $a \rightarrow b$ means that b catches a subset of the failures of a , and is therefore subsumed by a . Concretely, a may be a function that checks for the presence of gratitude-related keywords in the LLM response, and b may check for a subset of keywords that a checks. Here, if we were to minimize $\sum_j (x_j + s_j)$, we would simply pick a and e to be part of F' , since the rest of the functions would be subsumed (therefore their $s_j = 0$ —recall that $s_j = 1$ for assertions that are neither subsumed nor selected), and our overall metric would evaluate to 2. One intuitive way to view this problem is to start by keeping all of the nodes as unselected, i.e., $x_j = 0, s_j = 1$, and then by adding them one at a time in a predefined order to F' , we set $x_j = 1, s_j = 0$, and at the same time, we also impact s_j (moving them from 1 to 0) for all other functions that then become subsumed as a result.

More generally, subsumption relationships can be represented in the form of a Directed Acyclic Graph (DAG), as in our example above. (We omit the trivial case where two or more functions are equivalent, that is the only case where there can be cycles; in all other cases we have a DAG.) Within this DAG, we simply select all nodes that have no incoming edges and add them to F' . We can see that the rest of the nodes are subsumed, since they have at

least one incoming edge. We could certainly exclude a node that doesn’t have any incoming edge from F' , but adding it to F' does not worsen the objective because there is no way that that node will be subsumed by another. So, overall, we need to do a topological sort of the subsumption graph and pick the nodes “at the top”. Thus, the problem is in PTIME when there are no examples provided.

4 EVALUATION

We first discuss the LLM pipelines and datasets (i.e., E'); then, we discuss methods and metrics and present our results. The experiment code, datasets, and LLM responses are hosted on GitHub.

4.1 Pipeline and Dataset Descriptions

We evaluated SPADE on nine LLM pipelines—eight from LangChain Hub⁸, an open-source collection of LLM pipelines, and 1 proprietary pipeline. Each pipeline consists of a prompt template for the LLM and collection of approximately 75 examples (i.e., inputs and outputs in E') with labels for whether the outputs were good or bad. All nine pipelines come from LangChain users. Six pipelines were used in developing our prompt delta taxonomy (Section 2.2), each representing a different domain (e.g., programming, finance, marketing). Two pipelines came from SPADE’s Streamlit deployment (Section 2.4). We include one final proprietary pipeline (the *fashion* pipeline) in our experiments because it had the largest number of prompt template versions and SPADE’s assertions have already been deployed in production for tens of thousands of daily runs.

Table 3 provides details on each LLM pipeline and corresponding set of good and bad examples. For the *fashion* pipeline, good and bad examples were provided by a developer at the corresponding startup. While we used real user-provided prompt templates and histories (between 3 and 16 prompt versions) for the other 8 pipelines, we constructed and annotated our own input-output examples so we could release them publicly. For two pipelines, we sourced examples from Kaggle. For the other six pipelines, we synthetically generated the other datasets using Chat GPT Pro (based on GPT-4) and manually reviewed them. For instance, for the *codereviews* pipeline that uses an LLM to review pull requests, we asked Chat GPT to generate example pull requests covering a variety of programming languages, application types, and diff sizes. On average, we collected 75 examples per pipeline. We then executed the LLM pipelines on these inputs and manually labeled the responses to assess whether they met the prompt instructions.

4.2 Method Comparison and Metrics

As before, let E' be a dataset of example prompt-response pairs, as well as the corresponding labels of whether the response was good (i.e., 1) or bad (i.e., 0). Let τ be the FFR threshold and F be the set of candidate assertions produced by the first step of SPADE (Section 2). If a candidate function f results in a runtime error for some example e , we denote $f(e) = 0$ (i.e., failure). All code was written in Python, using PuLP to find solutions for the ILPs. We used the default PuLP configuration, which uses the CBC solver [17].

The simplest baseline involves generating candidate assertions and choosing all of them, but it proved ineffective, yielding 100% coverage and 100% FFR due to at least one assertion failing all tests.

⁸<https://smith.langchain.com/hub>

Table 3: Description of data-generating LLM pipelines in our experiments. The fashion examples (and ground-truth indicators of whether the example response is good or bad) are provided by a startup that uses LangChain. All other examples are synthetically generated except examples for the *finance* and *lecturesummaries* pipelines, which are taken from Kaggle.

Pipeline	# Good Ex.	# Bad Ex.	# Prompt Ver.	Data Generation Task	Full Prompt Link
<i>codereviews</i>	60	16	8	Writing reviews of GitHub repo pull requests	homanp/github-code-reviews
<i>emails</i>	43	55	3	Creating SaaS user onboarding emails	gitmaxd/onboard-email
<i>fashion</i>	48	34	16	Suggesting outfit ideas for specific events	N/A
<i>finance</i> ^a	48	52	5	Summarizing financial earnings call transcripts	casazza/map_template
<i>lecturesummaries</i> ^b	27	22	14	Summarizing lectures or talks, focusing on main points and critical insights	kirby/simple-lecture-summary
<i>negotiation</i>	27	19	8	Writing tailored negotiation strategies based on provided contracts and target prices	antoniogonc/strategy-report
<i>sportroutine</i>	19	31	3	Transforming workout video transcripts into structured exercise routines	aaalexlit/sport-routine-to-program
<i>statsbot</i>	39	31	3	Writing interactive discussions for any topic in statistics	anthonymolan/statistics-teacher
<i>threads</i>	50	56	4	Crafting concise, engaging Twitter threads for specific audiences and topics	fflo/summarization

^a <https://www.kaggle.com/datasets/ashwinm500/earnings-call-transcripts>

^b <https://www.kaggle.com/datasets/miguelcorraljr/ted-ultimate-dataset>

Table 4: Results of different versions of SPADE with $\alpha = 0.6$ and $\tau = 0.25$. “# CA” is short for the number of candidate assertions. The \checkmark and \times marks denote whether α and τ constraints are met. Each entry is a fraction of the total number of candidate assertions for that pipeline (with the absolute number in parentheses). SPADE_{cov} selects the fewest assertions overall. SPADE_{sub} selects the fewest assertions while optimizing for subsumption.

Pipeline	# CA	Method	FFR	Coverage on E'	Frac Func. Selected	Frac Excl. Funcs. not Subsumed
<i>codereviews</i>	44	BASLINE	0.117 \checkmark	1 \checkmark	0.456 (20)	0 (0)
		SPADE _{cov}	0 \checkmark	0.625 \checkmark	0.045 (2)	0.409 (18)
		SPADE _{sub}	0.117 \checkmark	0.875 \checkmark	0.341 (15)	0 (0)
<i>emails</i>	24	BASLINE	0 \checkmark	1 \checkmark	0.5 (12)	0 (0)
		SPADE _{cov}	0 \checkmark	1 \checkmark	0.0417 (1)	0.458 (11)
		SPADE _{sub}	0 \checkmark	1 \checkmark	0.458 (11)	0 (0)
<i>fashion</i>	106	BASLINE	0.878 \times	0.971 \checkmark	0.632 (67)	0 (0)
		SPADE _{cov}	0.245 \checkmark	0.6 \checkmark	0.028 (3)	0.321 (34)
		SPADE _{sub}	0.224 \checkmark	0.62 \checkmark	0.377 (40)	0 (0)
<i>finance</i>	47	BASLINE	0.667 \times	1 \checkmark	0.787 (37)	0 (0)
		SPADE _{cov}	0.229 \checkmark	0.673 \checkmark	0.085 (4)	0.553 (26)
		SPADE _{sub}	0.208 \checkmark	0.981 \checkmark	0.553 (26)	0 (0)
<i>lecturesum.</i>	70	BASLINE	0.528 \times	1 \checkmark	0.457 (32)	0 (0)
		SPADE _{cov}	0.194 \checkmark	0.643 \checkmark	0.014 (1)	0.414 (29)
		SPADE _{sub}	0.194 \checkmark	1 \checkmark	0.343 (24)	0 (0)
<i>negotiation</i>	50	BASLINE	0.444 \times	1 \checkmark	0.4 (20)	0 (0)
		SPADE _{cov}	0.222 \checkmark	0.632 \checkmark	0.04 (2)	0.32 (16)
		SPADE _{sub}	0.185 \checkmark	1 \checkmark	0.34 (17)	0 (0)
<i>sportroutine</i>	26	BASLINE	0.211 \checkmark	1 \checkmark	0.538 (14)	0 (0)
		SPADE _{cov}	0.211 \checkmark	0.774 \checkmark	0.077 (2)	0.462 (12)
		SPADE _{sub}	0 \checkmark	0.871 \checkmark	0.308 (8)	0 (0)
<i>statsbot</i>	15	BASLINE	0 \checkmark	1 \checkmark	0.467 (7)	0 (0)
		SPADE _{cov}	0 \checkmark	0.935 \checkmark	0.133 (2)	0.333 (5)
		SPADE _{sub}	0 \checkmark	1 \checkmark	0.467 (7)	0 (0)
<i>threads</i>	34	BASLINE	0 \checkmark	1 \checkmark	0.765 (26)	0 (0)
		SPADE _{cov}	0 \checkmark	0.875 \checkmark	0.029 (1)	0.735 (25)
		SPADE _{sub}	0 \checkmark	1 \checkmark	0.589 (20)	0 (0)

Failures were due to errors or overly specific conditions that, in theory, could pass some outputs (for example, requiring a precise phrase in a certain case) but, in reality, never matched any actual LLM outputs. As such, we evaluated two versions of SPADE against

a baseline that simply filters candidate assertions that individually exceed the FFR threshold. The BASELINE selects all functions f in F where $\text{FFR}(\{f\}) \leq \tau$. SPADE_{cov} is a solution to the ILP defined in Section 3.2 SPADE_{sub} is a solution to the ILP defined in Section 3.3.1.

Let F' represent the set of selected assertions by any version of SPADE. We measure four metrics:

- (1) Fraction of Assertions Selected (i.e., $|F'|/|F|$)
- (2) Fraction of Excluded Non-Subsumed Functions (i.e., $|G|/|F|$, where $G = \{g \mid g \in F \setminus F' \text{ and } F' \not\models g\}$)
- (3) False Failure Rate (Definition 3.2)
- (4) Coverage on E' (Definition 3.1)

An important aspect of SPADE_{sub}’s success is the effectiveness of subsumption assessment. Since we do not have ground truth for subsumption, we focus on precision, calculated as the proportion of *correctly* identified subsumed pairs of assertions out of all subsumed pairs identified by the LLM. We do not assess recall—whether GPT-4 identified every possible subsumption—due to the impracticality of labeling possibly tens of thousands of assertion pairs per pipeline. Moreover, precision is more critical than recall, as identifying even some subsumptions allows SPADE_{sub} to achieve a solution with fewer selected assertions.

4.3 Results and Discussion

Table 5: Precision of assessing subsumption with GPT-4 (verified by two authors).

Pipeline	Precision
<i>codereviews</i>	0.90
<i>emails</i>	0.79
<i>fashion</i>	0.74
<i>finance</i>	0.79
<i>lecturesummaries</i>	0.89
<i>negotiation</i>	0.68
<i>sportroutine</i>	0.89
<i>statsbot</i>	0.86
<i>threads</i>	0.80

Table 6: Average ILP runtimes (in seconds) over 10 trials for SPADE_{cov} and SPADE_{sub}. The BASELINE method has no ILP component.

Pipeline	SPADE _{cov}	SPADE _{sub}
<i>codereviews</i>	0.267	0.362
<i>emails</i>	0.196	0.225
<i>fashion</i>	0.628	1.197
<i>finance</i>	0.352	0.441
<i>lecturesummaries</i>	0.265	0.538
<i>negotiation</i>	0.220	0.332
<i>sportroutine</i>	0.120	0.158
<i>statsbot</i>	0.104	0.122
<i>threads</i>	0.272	0.332

First, we briefly discuss whether individual components of SPADE are practical (e.g., determining subsumption, solving the ILPs). Using GPT-4 to assess subsumption results in an average precision of 0.82 across all pipelines, as seen in Table 5, confirming its effectiveness. The ILP runtimes in Table 6 are all under one second, except for 1.197 seconds for SPADE_{sub} for the fashion pipeline. This relatively small runtime demonstrates the feasibility of our approach. In the remainder of this section, we will focus on the assertions chosen by different methods.

For simplicity, we set the coverage and FFR thresholds to be the same across all pipelines ($\alpha = 0.6$, $\tau = 0.25$). We report results for the three methods in Table 4. Consider the *codereviews* pipeline, for example, which uses an LLM to review a pull request for any code repository. Here, BASELINE selects 20 assertions, SPADE_{COV} selects two assertions, and SPADE_{sub} selects 15 assertions. By selecting more functions, SPADE_{sub} ensures that all non-subsumed functions are included. All three approaches respect the E' coverage constraint, but BASELINE violates the FFR constraint in 4 out of 9 pipelines. For our workloads, SPADE_{sub} opts for approximately 14% fewer assertions compared to BASELINE and shows a significantly lower FFR, reducing it by about 21% relative to BASELINE. SPADE_{COV} excludes, on average, about 44% of functions that are not subsumed by E' . Choosing a SPADE implementation primarily depends on how much labeled data is available. We subsequently discuss trade-offs between different SPADE implementations.

Subsumption vs. E' Coverage. SPADE_{COV} and SPADE_{sub} are complementary, the former being more useful if E' is more comprehensive. For our datasets, E' is not comprehensive: Table 4 reveals that, on average, 44% of functions excluded by SPADE_{COV} are not subsumed by the selected functions, despite being accurate within the FFR threshold. This is unsurprising given that each task has only 34 bad (i.e., failure) examples on average. While larger or more mature organizations may have extensive datasets and could get a meaningful result from SPADE_{COV}, SPADE_{sub}'s ability to select assertions that cover unrepresented *potential* failures can be beneficial in data-scarce settings. For example, here is a sample of 3 assertions for the *codereviews* pipeline ignored by SPADE_{COV} but included in SPADE_{sub} (with comments excluded for brevity):

```
async def assert_includes_code_improvement_v2(
    example: dict, prompt: str, response: str
):
    question = "Does the response include suggestions for code improvements?"
    return await ask_llm(prompt, response, question)

async def assert_contains_brief_answers_v1(example: dict, prompt: str,
    response: str):
    question = "Is the response brief and to the point without unnecessary
    elaboration?"
    return await ask_llm(prompt, response, question)

async def assert_responds_to_correct_pull_request(
    example: dict, prompt: str, response: str
):
    pr_title = example["title"]
    question = (
        f"Is the response a review focused on the Pull Request titled '{
        pr_title}'?"
    )
    return await ask_llm(prompt, response, question)
```

Subsumption as a Means for Reducing Redundancy. Several pipelines exhibit a large discrepancy between functions selected in

BASELINE and SPADE_{sub}, which occurs when there are many redundant candidates. For example, in the *codereviews* pipeline's 8 prompt versions, the developer iterated several times on the instruction to give a clear and concise review, resulting in five assertions that check the same thing (three of which are shown below):

```
async def assert_response_is_concise_v1(
    example: dict, prompt: str, response: str
) -> bool:
    question = "Is the LLM response concise and to the point?"
    return await ask_llm(prompt, response, question)

async def assert_response_is_concise_and_clear(
    example: dict, prompt: str, response: str
):
    question = "Is this pull request review response concise and clear?"
    return await ask_llm(prompt, response, question)

async def assert_clear_professional_language_v1(
    example: dict, prompt: str, response: str
):
    question = "Is the response professional, clear, and without unnecessary
    jargon or overly complex vocabulary?"
    return await ask_llm(prompt, response, question)
```

Since all the five assertions meet the FFR constraint, individually, BASELINE would select them all, which is undesirable because they all do the same thing, but SPADE_{sub} would select the one most compatible with the FFR constraint, as long as subsumption is assessed correctly. On the flip side, while assessing subsumption, the LLM may not recall all subsumptions, so SPADE_{sub} may have duplicate assertions. For example, the *codereviews* pipeline contains assertions titled `assert_includes_code_improvement_v1` and `assert_includes_code_improvement_v2`.

α and τ Threshold Sensitivity. The feasibility of solutions from the ILP solver in SPADE is dependent on the chosen α and τ thresholds. If a feasible solution is not found, developers may need to adjust these values in a binary search fashion. In our case, all 9 LLM pipelines yielded feasible solutions with $\alpha = 0.6$ and $\tau = 0.25$. However, the small size of E' makes SPADE_{COV} particularly sensitive to α . In the pipelines, we observed that between one and five assertions covered 60% of E' 's failures. For example, SPADE_{COV} selected only one assertion for the *emails* pipeline:

```
async def assert_encouragement_to_contact_company(
    example: dict, prompt: str, response: str
) -> bool:
    contact_phrases = [
        "reach out",
        "don't hesitate to contact",
        "looking forward to hearing from you",
        "if you have any questions",
        "need help getting started",
    ]
    return any(phrase in response for phrase in contact_phrases)
```

If E' is exhaustive of failure modes and representative of the distribution of failures (e.g., for the *emails* pipeline, most failures are actually due to the response lacking an encouragement to contact the company), SPADE_{COV} might be a satisfactory solution. However, our E' datasets clearly were not exhaustive, considering that SPADE_{sub} always chose additional assertions. SPADE_{sub} is less sensitive to α , as it explicitly selects assertions based on their potential to cover new failures (i.e., subsumption) without exceeding the FFR, even if the constraint on coverage is no longer tight.

FFR Tradeoffs. Considering that the difference between the fraction of functions selected for BASELINE and SPADE_{sub} is less than

10% for three LLM pipelines, one may wonder if the complexity of $\text{SPADE}_{\text{sub}}$ is worth it. $\text{SPADE}_{\text{sub}}$ is generally preferable because BASELINE fails to consistently meet the FFR threshold τ . We observed that as prompt versions increase, so do the number of assertions, impacting BASELINE adversely. The worst-case FFR of a set is the sum of individual FFRs, as shown in Equation (1). Hence, with a large number of independent assertions, the total FFR is likely to surpass the threshold. This issue is evident in the *fashion* and *lecturesummaries* pipelines, where despite each of the 67 and 32 assertions meeting FFR constraints individually, the total FFR for BASELINE reaches 88% and 53%, respectively. In practice, if SPADE were to be deployed in an interactive system, where SPADE could observe each LLM call in real-time (e.g., as a wrapper around the OpenAI API), the multitude of prompt versions further necessitates filtering assertions based on overall FFR. This underscores the need for the more complex $\text{SPADE}_{\text{cov}}$ or $\text{SPADE}_{\text{sub}}$ approaches.

5 RELATED WORK

We survey work from prompt engineering, evaluating ML and LLMs, LLMs for software testing, and testing ML pipelines.

Prompt Engineering. For both nontechnical [63] and technical users [37, 52], prompt engineering is hard for several reasons: small changes in prompt phrasing [3, 30] or the order of instructions or contexts [32] can significantly affect outputs. Moreover, as LLMs change under the hood of the API (i.e., prompt drift), outputs can change without developer awareness [9]. Tools and papers are emerging to aid in prompt management and experimentation, and are even using LLMs to write prompts [2, 11, 60, 61, 65]. Moreover, *deployed* prompts introduce new challenges, like “balancing more context with fewer tokens” and “wrangling prompt output” to meet user-defined criteria [38]. Our work doesn’t focus explicitly on helping developers create better prompts, but it could indirectly support developers by helping identify mistakes.

ML and LLM Evaluation. Evaluating and monitoring deployed ML models is known to be challenging [36, 49]. Evaluating LLMs in a deployed setting is even more challenging because LLMs are typically used for generative tasks, where the outputs are free-form [12]. Some LLM pipeline types, like question-answering with retrieval-augmented generation pipelines [29], can use standardized, automated metrics [13, 43], but others face challenges due to unknown metrics and lack of labeled datasets [8, 38, 62]. Typically, organizations rely on human evaluators for LLM outputs [18, 38, 58], but recent studies suggest LLMs can self-evaluate effectively with detailed “scorecards” [7, 26, 64]. However, writing these scorecards can be challenging [38], motivating auto-generated evaluators. Recent work [26, 41, 54] and industry tools [21, 27, 31] proposes the use of assertions to catch mistakes in LLM pipelines, while requiring the user to select these assertions.

LLMs for Software Testing. LLMs are increasingly being used in software testing, mainly for generating unit tests and test cases [28, 44, 53, 55, 56]. Research explores how LLMs’ prompting strategies, hallucinations, and nondeterminism affect code or test accuracy [10, 14, 15, 35]. Our work is complementary and leverages LLMs to generate code-based assertions for LLM pipelines.

Testing and Validation in ML Pipelines. ML pipelines are hard to manage in production. Much of the literature on ML testing

is geared towards validating structured data, through analyzing data quality [5, 22, 46, 47] or provenance [33, 45]. Platforms for ML testing typically offer automated experiment tracking and prevention against overfitting [1, 42], as well as tools for data distribution debugging [19]. Model-specific assertions typically require human specification [25], or at least large amounts of data to train learned assertions [24]. LLM chains or pipelines are a new class of ML pipelines, and LLMs themselves can generate assertions with little data. A recent study highlights the difficulty of testing LLM pipelines for “copilot”-like products: developers want to ensure accuracy while avoiding excessive resource use, such as running hundreds of assertions [38]—motivating assertion filtering.

6 CONCLUSION AND FUTURE WORK

We introduce a new problem of auto-generating assertions to catch failures in LLM pipelines, as well as SPADE , our framework for doing so. SPADE comprises two components: first, it synthesizes candidate assertions, and then it filters them down into a more manageable subset. To synthesize candidate assertions, we analyzed prompt version histories and learned that prompt deltas are often a rich source of requirements and therefore candidate assertions. We developed a taxonomy of prompt deltas for assertion synthesis, demonstrating its value via integration and deployment with LangChain, with over 2000 runs across domains. For the latter problem of candidate assertion filtering, we expressed the selection of an optimal set of assertions, covering most failures while introducing as few false failures as possible as an Integer Linear Program (ILP). We proposed assertion subsumption to cover failures in data-scarce scenarios and incorporated this into our ILP. We also studied the setting where there are no examples and demonstrated that it reduces to a topological sort on the subsumption graph. Our auto-generating assertion system, SPADE , was evaluated on nine real-world data-generating LLM pipelines. We have made our code and datasets public for further research and analysis.

There are a number of open questions in our effort to make production LLM pipelines more robust. For instance, while meeting developer-provided criteria (α , τ) is helpful, sometimes developers would like to examine the generated and selected assertions in a way that helps them make the tradeoffs themselves, motivating a human-in-the-loop interface to *assist* developers in defining data quality for LLM pipelines. Such an interface could also be a vehicle for getting developers to label examples on the fly. Determining which labeled examples would help best select from the set of assertions is an open question that is reminiscent of active learning. Finally, we could also envision automatically updating assertion sets in deployed pipelines, as new failure modes inevitably arise in production.

ACKNOWLEDGMENTS

We acknowledge support from the National Science Foundation (grants DGE-2243822, IIS-2129008, IIS-1940759, IIS-1940757, 1845638, 1740305, 2008295, 2106197, 2103794, 2312991), a BIDS Accenture Fellowship, funds from the State of California, an NDSEG Fellowship, funds from the Alfred P. Sloan Foundation, and EPIC lab sponsors (G-Research, Adobe, Microsoft, Google, Sigma Computing). Additional support was provided by Amazon and CAIT.

REFERENCES

- [1] Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, Vishrut Shah, Bocho Shen, Laura Sugden, Kaiyu Zhao, and Ming-Chuan Wu. 2019. Data Platform for Machine Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1803–1816. <https://doi.org/10.1145/3299869.3314050>
- [2] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena Glassman. 2023. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. *arXiv preprint arXiv:2309.09128* (2023).
- [3] Simran Arora, Avani Narayan, Mayee F Chen, Laurel Orr, Neel Guha, Kush Bhatia, Ines Chami, Frederic Sala, and Christopher Ré. 2022. Ask me anything: A simple strategy for prompting language models. *arXiv preprint arXiv:2210.02441* (2022).
- [4] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *MLSys*.
- [5] Eric Breck, Marty Zinkevich, Neoklis Polyzotis, Steven Whang, and Sudip Roy. 2019. Data Validation for Machine Learning. In *Proceedings of SysML*. <https://mlsys.org/Conferences/2019/doc/2019/167.pdf>
- [6] Rui Castro and Robert Nowak. 2008. Active learning and sampling. In *Foundations and Applications of Sensor Management*. Springer, 177–200.
- [7] Chi-Min Chan, Weize Chen, Yusheng Su, Jianxuan Yu, Wei Xue, Shanghang Zhang, Jie Fu, and Zhiyuan Liu. 2023. Chateval: Towards better llm-based evaluators through multi-agent debate. *arXiv preprint arXiv:2308.07201* (2023).
- [8] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Kaijie Zhu, Hao Chen, Linyi Yang, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, et al. 2023. A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109* (2023).
- [9] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. How is ChatGPT's behavior changing over time? *arXiv preprint arXiv:2307.09009* (2023).
- [10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [11] Yu Cheng, Jieshan Chen, Qing Huang, Zhenchang Xing, Xiwei Xu, and Qinghua Lu. 2023. Prompt Sapper: A LLM-Empowered Production Tool for Building AI Chains. *arXiv preprint arXiv:2306.12028* (2023).
- [12] Yao Dou, Maxwell Forbes, Rik Koncel-Kedziorski, Noah A Smith, and Yejin Choi. 2021. Is GPT-3 text indistinguishable from human text? SCARECROW: A framework for scrutinizing machine text. *arXiv preprint arXiv:2107.01294* (2021).
- [13] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2023. RA-GAS: Automated Evaluation of Retrieval Augmented Generation. *arXiv preprint arXiv:2309.15217* (2023).
- [14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. *arXiv preprint arXiv:2310.03533* (2023).
- [15] Robert Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards Autonomous Testing Agents via Conversational Large Language Models. *arXiv preprint arXiv:2306.05152* (2023).
- [16] Raul Castro Fernandez, Aaron J. Elmore, Michael J. Franklin, Sanjay Krishnan, and Chenhao Tan. 2023. How Large Language Models Will Disrupt Data Management. *Proc. VLDB Endow.* 16, 11 (jul 2023), 3302–3309. <https://doi.org/10.14778/3611479.3611527>
- [17] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. In *Emerging theory, methods, and applications*. INFORMS, 257–277.
- [18] Sebastian Gehrmann, Elizabeth Clark, and Thibault Sellam. 2023. Repairing the cracked foundation: A survey of obstacles in evaluation practices for generated text. *Journal of Artificial Intelligence Research* 77 (2023), 103–166.
- [19] Stefan Grafberger, Paul Groth, Julia Stoyanovich, and Sebastian Schelter. 2022. Data distribution debugging in machine learning pipelines. *The VLDB Journal* 31, 5 (2022), 1103–1126.
- [20] Madeleine Grunke-McLaughlin, Michelle S Lam, Ranjay Krishna, Daniel S Weld, and Jeffrey Heer. 2023. Designing LLM Chains by Adapting Techniques from Crowdsourcing Workflows. *arXiv preprint arXiv:2312.11681* (2023).
- [21] Guardrails 2023. Guardrails AI. <https://github.com/guardrails-ai/guardrails>.
- [22] Nick Hynes, D. Sculley, and Michael Terry. 2017. The Data Linter: Lightweight Automated Sanity Checking for ML Data Sets. http://learningsys.org/nips17/assets/papers/paper_19.pdf
- [23] Adam Tauman Kalai and Santosh S Vempala. 2023. Calibrated Language Models Must Hallucinate. *arXiv preprint arXiv:2311.14648* (2023).
- [24] Daniel Kang, Nikos Archiga, Sudeep Pillai, Peter D Bailis, and Matei Zaharia. 2022. Finding label and model errors in perception data with learned observation assertions. In *Proceedings of the 2022 International Conference on Management of Data*. 496–505.
- [25] Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2020. Model assertions for monitoring and improving ML models. *Proceedings of Machine Learning and Systems* 2 (2020), 481–496.
- [26] Tae Soo Kim, Yoonjoo Lee, Jamin Shin, Young-Ho Kim, and Juho Kim. 2023. EvallLM: Interactive Evaluation of Large Language Model Prompts on User-Defined Criteria. *arXiv preprint arXiv:2309.13633* (2023).
- [27] Langchain 2023. Langchain AI. <https://github.com/langchain-ai/langchain>.
- [28] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. CODAMOSA: Escaping coverage plateaus in test generation with pre-trained large language models. In *International conference on software engineering (ICSE)*.
- [29] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
- [30] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.
- [31] Llama Index 2023. Llama Index. https://github.com/run-llama/llama_index.
- [32] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. 2021. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786* (2021).
- [33] Mohammad Hossein Namaki, Avriella Floratu, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) (KDD '20). Association for Computing Machinery, New York, NY, USA, 1542–1551. <https://doi.org/10.1145/3394486.3403205>
- [34] Avani Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).
- [35] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).
- [36] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. 2022. Challenges in deploying machine learning: a survey of case studies. *Comput. Surveys* 55, 6 (2022), 1–29.
- [37] Aditya G Parameswaran, Shreya Shankar, Parth Asawa, Naman Jain, and Yujie Wang. 2023. Revisiting Prompt Engineering via Declarative Crowdsourcing. *arXiv preprint arXiv:2308.03854* (2023).
- [38] Chris Parnin, Gustavo Soares, Rahul Pandita, Sumit Gulwani, Jessica Rich, and Austin Z. Henley. 2023. Building Your Own Product Copilot: Challenges, Opportunities, and Needs. *arXiv:2312.14231* [cs.SE]
- [39] Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2018. Data lifecycle challenges in production machine learning: a survey. *ACM SIGMOD Record* 47, 2 (2018), 17–28.
- [40] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2020. Snorkel: Rapid training data creation with weak supervision. *The VLDB Journal* 29, 2-3 (2020), 709–730.
- [41] Traian Rebedea, Razvan Dinu, Makesh Sreedhar, Christopher Parisien, and Jonathan Cohen. 2023. Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails. *arXiv preprint arXiv:2310.10501* (2023).
- [42] Cedric Renggli, Frances Ann Hubis, Bojan Karlas, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2019. Ease.MI/Ci and Ease.MI/Meter in Action: Towards Data Management for Statistical Generalization. *Proc. VLDB Endow.* 12, 12 (aug 2019), 1962–1965. <https://doi.org/10.14778/3352063.3352110>
- [43] Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. 2023. ARES: An Automated Evaluation Framework for Retrieval-Augmented Generation Systems. *arXiv preprint arXiv:2311.09476* (2023).
- [44] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *arXiv preprint arXiv:2302.06527* (2023).
- [45] Sebastian Schelter, Stefan Grafberger, Shubha Guha, Bojan Karlas, and Ce Zhang. 2023. Proactively Screening Machine Learning Pipelines with ARGUSEYES. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) (SIGMOD '23). Association for Computing Machinery, New York, NY, USA, 91–94. <https://doi.org/10.1145/3555041.3589682>
- [46] Sebastian Schelter, Dustin Lange, Philipp Schmidt, Meltem Celikel, Felix Biessmann, and Andreas Grafberger. 2018. Automating Large-Scale Data Quality Verification. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1781–1794. <https://doi.org/10.14778/3229863.3229867>
- [47] Shreya Shankar, Labib Fawaz, Karl Gyllstrom, and Aditya Parameswaran. 2023. Automatic and Precise Data Validation for Machine Learning. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. 2198–2207.
- [48] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. 2022. Operationalizing machine learning: An interview study. *arXiv preprint arXiv:2209.09125* (2022).
- [49] Shreya Shankar, Bernease Herman, and Aditya G Parameswaran. 2022. Rethinking streaming machine learning evaluation. *arXiv preprint arXiv:2205.11473* (2022).

- [50] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, JD Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G Parameswaran, and Eugene Wu. 2024. SPADE: Synthesizing Assertions for Large Language Model Pipelines. *arXiv preprint arXiv:2401.03038* (2024).
- [51] Shreya Shankar, Haotian Li, Will Fu-Hinthorn, Harrison Chase, J.D. Zamfirescu-Pereira, Yiming Lin, Sam Noyes, Eugene Wu, and Aditya Parameswaran. 2023. SPADE: Automatically digging up evals based on prompt refinements. <https://blog.langchain.dev/spade-automatically-digging-up-evals-based-on-prompt-refinements/>
- [52] Chenglei Si, Zhe Gan, Zhengyuan Yang, Shuohang Wang, Jianfeng Wang, Jordan Boyd-Graber, and Lijuan Wang. 2022. Prompting gpt-3 to be reliable. *arXiv preprint arXiv:2210.09150* (2022).
- [53] Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. 2023. Exploring the Effectiveness of Large Language Models in Generating Unit Tests. *arXiv preprint arXiv:2305.00418* (2023).
- [54] Arnav Singhvi, Manish Shetty, Shangyin Tan, Christopher Potts, Koushik Sen, Matei Zaharia, and Omar Khattab. 2023. DSPy Assertions: Computational Constraints for Self-Refining Language Model Pipelines. *arXiv preprint arXiv:2312.13382* (2023).
- [55] Benjamin Steenhoeck, Michele Tufano, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. Reinforcement Learning from Automatic Feedback for High-Quality Unit Test Generation. *arXiv preprint arXiv:2310.02368* (2023).
- [56] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2023. Software testing with large language model: Survey, landscape, and vision. *arXiv preprint arXiv:2307.07221* (2023).
- [57] Yidong Wang, Zhuohao Yu, Zhengran Zeng, Linyi Yang, Wenjin Yao, Cunxiang Wang, Hao Chen, Chaoya Jiang, Rui Xie, Jindong Wang, et al. 2023. PandaLM: An Automatic Evaluation Benchmark for LLM Instruction Tuning Optimization. In *The Twelfth International Conference on Learning Representations*.
- [58] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. 2023. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966* (2023).
- [59] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.
- [60] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. 2022. Promptchainer: Chaining large language model prompts through visual programming. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–10.
- [61] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–22.
- [62] Ziang Xiao, Susu Zhang, Vivian Lai, and Q Vera Liao. 2023. Evaluating NLG Evaluation Metrics: A Measurement Theory Perspective. *arXiv preprint arXiv:2305.14889* (2023).
- [63] JD Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–21.
- [64] Xinghua Zhang, Bowen Yu, Haiyang Yu, Yangyu Lv, Tingwen Liu, Fei Huang, Hongbo Xu, and Yongbin Li. 2023. Wider and deeper llm networks are fairer llm evaluators. *arXiv preprint arXiv:2308.01862* (2023).
- [65] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2022. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910* (2022).