

SPADE: Synthesizing 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

Operationalizing large language models (LLMs) for custom, repetitive data pipelines is challenging, particularly due to their unpredictable and potentially catastrophic failures. Acknowledging the inevitability of these errors, we focus on identifying *when* LLMs may be generating incorrect responses when used repeatedly as part of data generation pipelines. We present SPADE, a method for automatically synthesizing assertions that identify bad LLM outputs. SPADE analyzes prompt version histories 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.

1 INTRODUCTION

There is a lot of excitement around the use of large language models (LLMs) in data pipelines [17]. Much of the enthusiasm is due to their simplicity: without needing large labeled datasets, one can easily create an automated, intelligent pipeline that executes any arbitrary data generation task within seconds—simply by *prompting* an LLM with natural language. However, operationalizing LLMs for unsupervised, repeated, or large-scale data generation tasks presents significant challenges [34], such as potential errors, inappropriate responses, or hallucinations [44, 57].

Consider a movie streaming platform using LLMs to generate personalized movie recommendation notes. A developer might write a prompt template like: “write a personalized note for why a user should watch {movie_name} given the following information about the user: {personal_info}” to be executed for multiple user-movie pairs. The templated variables represent information that would get injected into the pipeline at runtime, allowing the pipeline to generate data for a variety of inputs. In theory, this prompt seems adequate, but the developer might observe some issues while testing it on a few example inputs: the LLM might reference a movie the user never watched, cite a sensitive attribute (e.g., race or ethnicity), or even have a basic issue like an overly short response.

To deal with such shortcomings, frameworks incorporating assertions into LLM pipelines have been proposed to filter out bad generations before reaching end-users [37, 49]. Given the unavoidable nature of LLM errors [23], such assertions are key to successful deployments of LLM pipelines that generate data. However, developers find it tough to write assertions for custom LLM pipelines [35]. Challenges include predicting all possible LLM failure modes, the

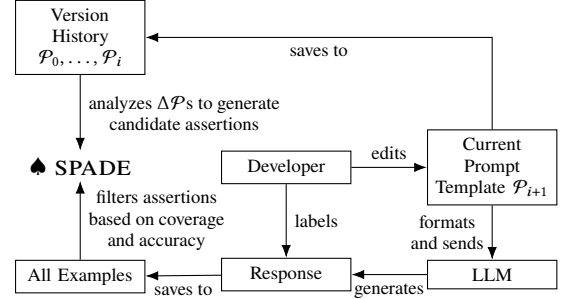


Figure 1: Before a developer deploys a prompt template to production, SPADE analyzes the deltas (i.e., diffs) between consecutive prompt templates to generate assertions and uses a handful of labeled responses to filter out redundant and inaccurate assertions while maintaining coverage.

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, 57]. Moreover, if there are too many or non-informative assertions, developers can be overwhelmed monitoring their results.

In this paper, we identify a new problem of automatically generating a good set of assertions for any data generation pipeline that leverages LLMs. While an assertion returns true (i.e., success) or false (i.e., failure) for an LLM response, a set of assertions returns the conjunction of individual assertions. A “good” set of assertions has minimal overlap and allows the developer to trade off between false successes and false failures. We decompose the problem into two components—candidate assertion generation and filtering—and present SPADE (System for **P**rompt **A**nalysis and **D**elta-Based **E**valuation, Figure 1). For candidate assertion generation, one may consider directly querying an LLM to “write assertions for x prompt,” but this may not cover the criteria the developer wants. We propose generating candidates from prompt deltas (i.e., diffs between two consecutive prompt versions), which often indicate specific failure modes of LLMs. For example, a developer might add an instruction like “avoid flowery language,” motivating an assertion to check the response language. We present an analysis of 19 custom data generation pipelines from users from LangChain (a startup that helps people build LLM pipelines) and pipeline prompt version histories, forming a taxonomy of prompt deltas. SPADE first

automatically categorizes prompt deltas within the taxonomy, then synthesizes Python functions (that may include LLM calls) as candidate assertions. We demonstrate the potential of these assertions with a public release of this component of SPADE, with over 1300 uses across more than 10 sectors like finance, medicine, and IT [46].

Our analysis of candidate assertions revealed redundancy, inaccuracy, and a large number of functions, often exceeding 50 for just a few prompt deltas. Redundancy stems from repeated refinements to similar portions of a prompt or prompts with ambiguous instructions (e.g., “return a concise response”). Reducing this redundancy is not straightforward, even for engineers, since assertions can involve LLM calls with varying accuracies, motivating an automated filtering component. One approach is to use developer-labeled LLM responses to estimate each assertion’s false failure rate (FFR) and eliminate each one exceeding a developer-defined FFR threshold. However, the remaining assertions can have an FFR that cumulatively exceeds this threshold, 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. However, we find that labeled responses may not represent all failure modes, leading to omission of valuable assertions. For instance, in our movie recommendation scenario, an assertion that correctly verifies if the LLM-generated note is under 200 words will get discarded if all responses in our developer-labeled sample are below this limit. To expand coverage, active learning and weak supervision approaches can be used to sample and label new LLM prompt-response pairs for each candidate assertion [6, 36], but this may be expensive or inaccessible for non-programmers. We introduce assertion *subsumption* as a way of ensuring comprehensive coverage: if one assertion doesn’t encompass the failure modes of another, both are selected. As such, SPADE selects a minimal set of assertions that respects failure coverage, accuracy, and subsumption constraints. Overall, we make the following contributions:

Prompt Deltas for Candidate Assertion Generation (Section 2). We identify prompt version history as a rich source of correctness criteria for data-generating LLM pipelines. We infer a taxonomy of assertion criteria based on prompt deltas from 19 LLM pipelines with version history, which we publish. We publicly released a tool¹ to generate candidate assertions for any pipeline, providing early evidence for the utility of auto-generated assertions with over 1300 deployments while identifying opportunities to filter these assertions.

Filtering Method that Requires Little Data (Section 3). We show that selecting a minimal set of assertions while covering failure modes and meeting accuracy requirements is NP-hard. We express this problem as an ILP. Given that assertions may be redundant and inaccurate, and that LLM developers may not have engineering expertise or enough data to use existing data-driven approaches to ML pipeline testing (e.g., training models), we introduce assertion subsumption as a novel proxy for coverage in low-data settings.

Empirical Study (Section 4). We present SPADE, our system that automatically generates assertions for data-generating LLM pipelines. For nine LLM pipelines with prompt version history, we collect labeled prompt-response pairs (eight of which we open-source).

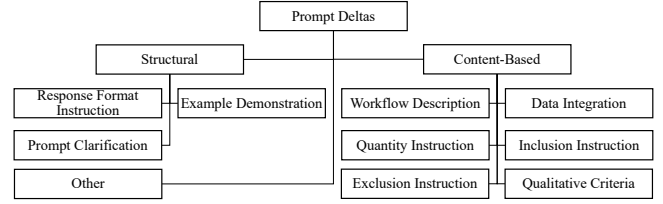


Figure 2: Taxonomy of prompt deltas from 19 LLM pipelines.

SPADE² generates assertions with good failure coverage and few false failures (i.e., failing a good response) across all pipelines. Our subsumption-based ILP outperforms simpler baselines that do not consider interactions between assertions by reducing assertions by 14% and lowering the false failure rate by 21%.

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 an input tuple t to derive a prompt p , and returns a response r . 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 Section 1’s movie recommendation examples, suppose there are 7 versions, where \mathcal{P}_7 is the following: “Write a personalized note for why a user should watch {movie_name} given the following information about the user: {personal_info}. 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* 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., “-”). For example, Table 1 shows the $\Delta\mathcal{P}$ s for a number of versions. 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—for example, $\Delta\mathcal{P}_6$ in Table 1 contains some new instructions added to a sentence from \mathcal{P}_5 . 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

We analyzed 19 LLM pipelines collected from LangChain users, each of which consists of between 3 and 11 historical prompt template versions. These pipelines span various tasks, from generating workout summaries to custom question-and-answering chatbots. Table 6 in Appendix B shows a summary of the pipelines, including a description

¹<https://spade-beta.streamlit.app>

²<https://github.com/shreyashankar/spade-experiments>

Version i	$\Delta\mathcal{P}_i$	Possible New Assertion Criteria
1	+ Write a personalized note for why a user should watch {movie_name} given the following information about the user: {personal_info}.	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.	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.	Response should be concise
4	- Ensure the recommendation note is concise. + Ensure the recommendation note is concise, not exceeding 100 words.	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.	Response should mention genre and verify cast members are accurate
6	+ Mention any awards or critical acclaim received by movie_name.	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.	Response should not include references to sensitive personal attributes

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} .

of each pipeline and the number of prompt versions. 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). We iterated on our categories 4 times, ultimately producing the taxonomy in Figure 2. The taxonomy-annotated dataset of prompt versions can be found online³. Most prompt deltas fell under Data Integration (i.e., adding placeholder variables) and Workflow Description (i.e., adding instructions for the LLM to perform the task more accurately). In Table 2, we show sample prompt deltas for each category in our taxonomy, using the movie recommendation pipeline example from the introduction.

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	Advices on omissions.	+ “Avoid movie plot spoilers.”
Qualitative Criteria	Sets stylistic attributes.	+ “Maintain friendly, positive tone.”

Table 2: Categories of prompt deltas.

When deriving assertion candidates from automatically-identified prompt deltas, assertion quality clearly depends on the accuracy of the identified categories. Therefore, we confirmed GPT-4’s correct categorization of prompt deltas (as of October 2023). We assigned ground truth categories to all prompt versions 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 Appendix C.

2.3 From Taxonomy to Assertions

Using our taxonomy, SPADE employs an LLM to categorize and generate assertions for prompt deltas, detailed in Appendix C. For each $\Delta\mathcal{P}_i$, SPADE prompts the LLM to suggest as many criteria as possible for assertions, each aligning with a taxonomy category

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

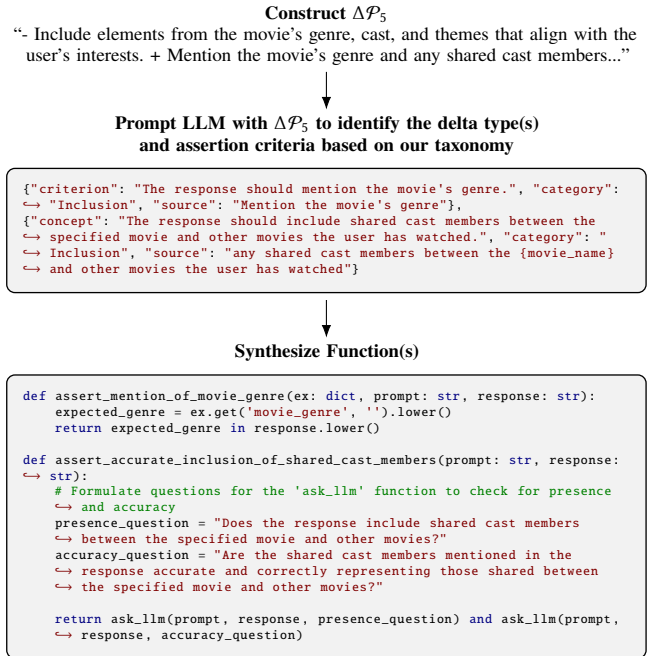


Figure 3: Generating candidate assertion functions from a $\Delta\mathcal{P}$.

(e.g., Figure 3). A criterion is loosely defined as some natural language expression that operates on an example and evaluates to True or False (e.g., “check for conciseness”). SPADE analyzes every $\Delta\mathcal{P}_i$ instead of just last prompt version for several reasons: developers often remove instructions from prompts to reduce costs while expecting the same behavior [35], 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.

For each delta, SPADE collects the criteria identified and prompts the 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 step is a multiset of candidate functions $F = \{f_1, \dots, f_m\}$.

We adopt a two-step process in our approach, as it has been demonstrated that breaking tasks into steps can enhance LLM accuracy [21, 53, 55]. However, it’s worth noting that with daily advancements in LLMs, a one-step process might already be feasible. Additionally, while our taxonomy guides LLM-generated assertions now, future LLMs may implicitly learn these categories through reinforcement learning from human feedback [12]. Nevertheless, knowing the taxonomy-based categories associated with candidate assertions may help in filtering them.

2.4 LangChain Deployment

To assess the potential of our auto-generated assertions, in the first week of November 2023, we released an early prototype of SPADE’s candidate assertions via a Streamlit application⁴. At the time of public release, the Streamlit application had a different set of prompts to generate assertions; these prompts generated relatively few assertions compared to the number of assertions generated in this paper’s version of SPADE. However, the taxonomy has remained the same. 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 automatically contains prompt version history 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.

We found significant interest in auto-generated LLM pipeline assertions: there were *over 1300 runs* of the app for custom prompt templates (i.e., not the sample default prompt template in app) within two weeks. These runs span a variety of fields, including medicine, education, cooking, and finance (Figure 5). Users for 10 of the 1300+ runs gave an optional “thumbs up” feedback on the assertions generated for them, and users for 43 of these runs clicked the “download assertions” button, which downloads the candidate assertions as a Python file. We note that users can copy the code in the assertions instead of downloading the assertions as a Python file, and we were unable to measure the copy events. No users clicked the “thumbs down” button—which is not to say that the generated assertions were perfect. In fact, anecdotally, we have seen the candidate assertion quality improve over the last several months as GPT-4 continually improves. On average, 3.3 assertions were generated per run, with minimum 1 and maximum 10 assertions generated for a run. Most of the runs corresponded to a single prompt version. We also found a deployment of SPADE-generated assertions from a LangChain user who built a “chat-with-your-pdf” tool⁵: in their words, “*When I saw it I didnt beleive it could work that well, but it really did and made the evaluation process fun and ez.*”

⁴<https://spade-beta.streamlit.app/>

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

Get Suggested Evaluation Functions

SPADE (System for Prompt Analysis and Delta-based Evaluation) will suggest binary eval functions for your prompt that you can run on all future LLM responses on. It works best when given the version history of your prompt (i.e., through a LangSmith Hub repo), but you can also use it with a single prompt template.

This is an experimental version of SPADE, built in collaboration with UC Berkeley and used for research purposes. [Here's a link](#) to our blog post. If you'd like to provide feedback on the quality of evals or participate in an interactive prompt engineering study so we can improve the tool, please fill out [this form](#).

How it works

Choose Input Type

Prompt Template

Prompt Template

A client ({client_genders}) wants to be styled for {event}. Suggest 5 apparel items for {client_pronoun} to wear. For wedding-related events, don't suggest any white items unless the client explicitly states that they want to be styled for their wedding. Return your answer as a python list of strings

Prompt Versions

Eval function generation in progress.

See In-Progress Analysis

Annotated first prompt template

A client ({client_genders}) wants to be styled for {event}. Suggest 5 apparel items for {client_pronoun} to wear. For wedding-related events, don't suggest any white items unless the client explicitly states that they want to be styled for their wedding. Return your answer as a python list of strings

Prompt refinement legend

- FormatInstruction
- ExampleDemonstration
- WorkflowDescription
- QuantityInstruction
- Inclusion
- Exclusion
- QualitativeAssessment

Suggested evaluation functions

```
# Needs LLM: False
def evaluate_python_list_format(prompt: str, response: str) -> bool:
    """
```

Figure 4: Screenshot of an early version of the SPADE Streamlit application.

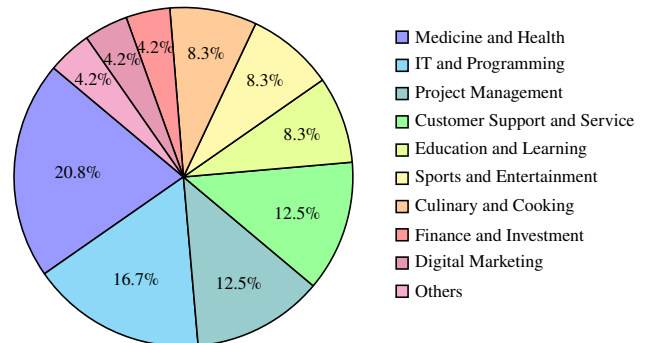


Figure 5: Prompts submitted to the SPADE Streamlit application span a variety of fields.

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 transcripts⁶ 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 summary’s 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 complex or LLM-invoking ones, where even experienced developers might not be able to gauge their effectiveness without viewing the results of the functions on many examples. 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.

3 FILTERING CANDIDATE ASSERTIONS

Here, we address the issue of redundant and incorrect assertions identified in Section 2.4, particularly in pipelines with numerous prompt versions. Filtering this candidate set not only improves efficiency when deploying the assertions to run in production, but it also reduces cognitive overhead for the developer.

3.1 Definitions

Consider e_i as an example end-to-end execution (i.e., run) of an LLM pipeline on some input. 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 success and 0 indicates failure. Let $F' = \{f_1, f_2, \dots, f_k\}$ be a set of k assertions. An example e_i is deemed *successful* by this set if and only if it satisfies all assertions in F' . Specifically:

$$\hat{y}_i = \begin{cases} 1 & \text{if } f(e_i) = 1 \quad \forall f \in F', \\ 0 & \text{otherwise.} \end{cases}$$

We denote $y_i \in \{0, 1\}$ to represent whether the LLM chain developer considers e_i to be a success (1) or failure (0). 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:

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

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:

$$\text{FFR}(F') = \frac{\sum_i \mathbb{I}[\hat{y}_i = 0 \wedge y_i = 1]}{\sum_i \mathbb{I}[y_i = 1]}$$

In both Definition 3.1 and Definition 3.2, \hat{y}_i represents set F' ’s prediction for the i -th example, y_i is the actual outcome determined by the LLM chain developer, while \mathbb{I} represents the indicator function. In practice, Definition 3.1 and Definition 3.2 are impossible to compute since the universe of examples E is unknown. So, for now, we assume access only 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 hypothetically 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. Table 7 in Appendix A summarizes the notation used throughout this section.

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 failure while $\alpha = 1$. 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 $f_j(e_i) = 1$ 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:

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

Then, the coverage constraint can be written as:

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

Now, we formulate the FFR constraint. Observe that FFR can be decomposed into the following:

$$\frac{\sum_{i=1}^n y_i \cdot \max_j (w_{ij})}{\sum_{i=1}^n [y_i = 1]} \leq \tau. \quad (1)$$

The product $w_{ij} = (1 - M_{ij}) \cdot x_j$ in the numerator is 1 if f_j is included in F' and incorrectly marks a good example as a failure. The max function ensures that as long as there is some selected

⁶<https://smith.langchain.com/hub/kirby/simple-lecture-summary>

function f_j that marks e_i as a failure, the product is 1. Multiplying y_i by the max result ensures that the numerator is incremented only when $y_i = 1$, or when the example e_i is actually not a failure. This numerator is thus equivalent to the numerator in Definition 3.2, i.e., $y_i = 1 \wedge \hat{y}_i = 0$. The entire fraction is less than the threshold τ , which represents the maximum allowed false failure rate across the examples. To formulate FFR according to (1), 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):

$$z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n]; \forall j \in [1, m].$$

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):

$$\begin{aligned} & \text{minimize} \quad \sum_{j=1}^m x_j \\ & \text{subject to:} \quad w_{ij} = (1 - M_{ij}) \cdot x_j, \quad \forall i \in [1, n], \forall j \in [1, m]; \\ & \quad u_i \leq \sum_{j=1}^m w_{ij}, \quad \forall i \in [1, n] \text{ where } y_i = 0; \quad \frac{\sum_{i: y_i=0} u_i}{\sum_i \mathbb{I}[y_i = 0]} \geq \alpha; \\ & \quad z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n], \forall j \in [1, m]; \quad \frac{\sum_{i=1}^n z_i}{\sum_{i=1}^n \mathbb{I}[y_i = 1]} \leq \tau. \end{aligned}$$

We refer to a solution for this ILP as $\text{SPADE}_{\text{cov}}$. 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.

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 , $\text{SPADE}_{\text{cov}}$ 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 [36]. 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 *subsumption*. 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.

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' :

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

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

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' .

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

Then, s_j denotes if f_j neither in F' nor subsumed by any function in F' :

$$\begin{aligned} s_j &\leq 1 - x_j, & s_j &\leq 1 - r_j, & \forall j \in [1, m]; \\ s_j &\geq (1 - x_j) + (1 - r_j) - 1 = 1 - x_j - r_j, & \forall j \in [1, m]. \end{aligned}$$

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):

$$\begin{aligned} & \text{minimize} \quad \sum_{j=1}^m x_j + \sum_{j=1}^m s_j \\ & \text{subject to:} \quad w_{ij} = (1 - M_{ij}) \cdot x_j, \quad \forall i \in [1, n], \forall j \in [1, m]; \\ & \quad u_i \leq \sum_{j=1}^m w_{ij}, \quad \forall i \in [1, n] \text{ where } y_i = 0; \quad \frac{\sum_{i: y_i=0} u_i}{\sum_i \mathbb{I}[y_i = 0]} \geq \alpha; \\ & \quad z_i \geq y_i \cdot w_{ij}, \quad \forall i \in [1, n], \forall j \in [1, m]; \quad \frac{\sum_{i=1}^n z_i}{\sum_{i=1}^n \mathbb{I}[y_i = 1]} \leq \tau; \\ & \quad r_j \leq \sum_{\substack{i=1 \\ i \neq j}}^m (x_i \cdot K_{ij}), \quad \forall j \in [1, m]; \\ & \quad s_j \leq 1 - x_j, \quad s_j \leq 1 - r_j, \quad \forall j \in [1, m]; \\ & \quad s_j \geq 1 - x_j - r_j, \quad \forall j \in [1, m]. \end{aligned}$$

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 calls to the

LLM 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 it can identify, then prompting it again to transform its response into a parse-able list of pairs $a \implies b$. Details of this LLM subsumption prompt are in Appendix C.

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 $f \notin S$,

$$\begin{aligned} \max(\text{FFR}(S), \text{FFR}(\{f\})) &\leq \text{FFR}(S \cup \{f\}), \\ \text{FFR}(S \cup \{f\}) &\leq \text{FFR}(S) + \text{FFR}(\{f\}). \end{aligned} \quad (2)$$

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

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 GitHub⁷.

4.1 Pipeline and Dataset Descriptions

We evaluate SPADE on nine LLM pipelines—eight from LangChain Hub⁸, an open-source collection of LLM pipelines, and 1 proprietary pipeline. Six LangChain Hub pipelines helped develop the prompt delta taxonomy (Section 2.2), but two pipelines were added from SPADE’s Streamlit deployment (Section 2.4) after the taxonomy was created. The proprietary pipeline is the *fashion* pipeline, which suggests outfits for events. This pipeline is included due to its use of data not in LLM training and its real-world deployment—demonstrating SPADE’s practical applicability.

While we used real user prompt templates and histories (between 3 and 16 prompt versions), we constructed our own sets of examples prompt-response pairs and labels (E') for testing. Two datasets for the LangChain Hub pipelines were sourced from Kaggle, while the others were synthetically generated using Chat GPT Pro (based on GPT-4). For instance, for the *codereviews* pipeline that uses an LLM to review pull requests, we asked Chat GPT to generate placeholder values covering a variety of programming languages, application types, and diff sizes. We labeled the LLM responses for the 8 LangChain pipelines to assess whether they met the prompt instructions. The responses were split between GPT-3.5-Turbo and GPT-4. For the *fashion* pipeline, labeling was done by a developer at the corresponding startup. Table 3 provides details on each LLM pipeline and dataset. We have open-sourced all data for the 8 LangChain Hub pipelines.

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 of our code was written in Python, using the PuLP Python package to find solutions for the ILPs. We used the default PuLP configuration, which uses the CBC solver [18]. We evaluated three versions of SPADE:

- SPADE_{base} 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 $G = \{g \mid g \in F \setminus F' \text{ and } F' \not\Rightarrow g\}$)
- (3) False Failure Rate (Definition 3.2)
- (4) Coverage on E' (Definition 3.1)

Additionally, an important aspect of SPADE_{sub}’s success is the effectiveness of subsumption assessment between all pairs of assertions. Since we do not have ground truth for subsumption, we focus on precision, calculated as the proportion of *correctly* identified subsumed pairs 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 or accuracy, as identifying even some subsumptions allows SPADE_{sub} to achieve a solution with fewer selected assertions than SPADE_{base}.

4.3 Results and Discussion

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. 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, SPADE_{base} selects 20 assertions, SPADE_{base} 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 SPADE_{base} violates the FFR constraint in 4 out of 9 pipelines.

On average, SPADE_{sub} opts for approximately 14% fewer assertions compared to SPADE_{base} and shows a significantly lower FFR, reducing it by about 21% relative to SPADE_{base}. SPADE_{cov} excludes, on average, about 44% of functions that are not subsumed by F' . We subsequently discuss the 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 definitely 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

⁷<https://github.com/shreyashankar/spade-experiments>

⁸<https://smith.langchain.com/hub>

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	homamp/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	antonigonc/strategy-report
<i>sportroutine</i>	19	31	3	Transforming workout video transcripts into structured exercise routines	aaalexli/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 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	# Candidate Assertions	Method	FFR	Coverage on E'	Frac Func. Selected	Frac Excl. Func. not Subsumed
<i>codereviews</i>	44	SPADE _{base}	0.117	✓	1	✓ 0.456 (20)
		SPADE _{cov}	0	✓	0.625	✓ 0.045 (2)
		SPADE _{sub}	0.117	✓	0.875	✓ 0.341 (15)
<i>emails</i>	24	SPADE _{base}	0	✓	1	✓ 0.5 (12)
		SPADE _{cov}	0	✓	1	✓ 0.0417 (1)
		SPADE _{sub}	0	✓	1	✓ 0.458 (11)
<i>fashion</i>	106	SPADE _{base}	0.878	✗	0.971	✓ 0.632 (67)
		SPADE _{cov}	0.245	✓	0.6	✓ 0.028 (3)
		SPADE _{sub}	0.224	✓	0.62	✓ 0.377 (40)
<i>finance</i>	47	SPADE _{base}	0.667	✗	1	✓ 0.787 (37)
		SPADE _{cov}	0.229	✓	0.673	✓ 0.085 (4)
		SPADE _{sub}	0.208	✓	0.981	✓ 0.553 (26)
<i>lecturesummaries</i>	70	SPADE _{base}	0.528	✗	1	✓ 0.457 (32)
		SPADE _{cov}	0.194	✓	0.643	✓ 0.014 (1)
		SPADE _{sub}	0.194	✓	1	✓ 0.343 (24)
<i>negotiation</i>	50	SPADE _{base}	0.444	✗	1	✓ 0.4 (20)
		SPADE _{cov}	0.222	✓	0.632	✓ 0.04 (2)
		SPADE _{sub}	0.185	✓	1	✓ 0.34 (17)
<i>sportroutine</i>	26	SPADE _{base}	0.211	✓	1	✓ 0.538 (14)
		SPADE _{cov}	0.211	✓	0.774	✓ 0.077 (2)
		SPADE _{sub}	0	✓	0.871	✓ 0.308 (8)
<i>statsbot</i>	15	SPADE _{base}	0	✓	1	✓ 0.467 (7)
		SPADE _{cov}	0	✓	0.935	✓ 0.133 (2)
		SPADE _{sub}	0	✓	1	✓ 0.467 (7)
<i>threads</i>	34	SPADE _{base}	0	✓	1	✓ 0.765 (26)
		SPADE _{cov}	0	✓	0.875	✓ 0.029 (1)
		SPADE _{sub}	0	✓	1	✓ 0.589 (20)

Table 4: Results of different versions of SPADE with $\alpha = 0.6$ and $\tau = 0.25$. The check and x-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	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 5: Precision of assessing subsumption with GPT-4. Two authors verified the results.

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 SPADE_{base} 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 (two 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, SPADE_{base} 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 SPADE_{BASE} and SPADE_{SUB} is less than 10% for three LLM pipelines, one may wonder if the complexity of SPADE_{SUB} is worth it. SPADE_{SUB} is generally preferable because SPADE_{BASE} fails to consistently meet the FFR threshold τ . We observed that as prompt versions increase, so do the number of assertions, impacting SPADE_{BASE} adversely. The worst-case FFR of a set is the sum of individual FFRs, as shown in Equation (2). 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 SPADE_{BASE} reaches 0.88 and 0.53, 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 SPADE_{COV} or SPADE_{SUB} approaches.

4.4 Limitations and Future Work

Improving Quality of LLMs. While LLMs (both closed and open-source) are improving rapidly, and we don't explicitly study prompt engineering strategies for SPADE, a complementary research idea is to explore such strategies or fine-tune small open-source models to generate assertions. Moreover, we proposed subsumption as a coverage proxy but didn't explore prompt engineering or even non-LLM strategies (e.g., assertion provenance) for assessing subsumption. Despite LLM advancements, SPADE's filtering stage remains crucial

for reducing redundancy and ensuring accuracy, especially since assertions may involve LLMs.

Collecting Labeled Examples. Acquiring labeled data (E') is hard. Most of our datasets had very few prompt versions (only the ones committed to LangChain Hub), but in reality, developers may iterate on their prompt tens or hundreds of times. Future work could involve passive example collection via LLM API wrappers or gathering developer feedback on assertions. Prioritizing different types of assertions and formalizing these priorities within SPADE is another area for exploration. Additionally, assessing the accuracy of FFR estimates with limited E' and exploring methods to enhance FFR accuracy in the absence of large labeled datasets (e.g., via prediction-powered inference [21]), presents an interesting area for future work.

Supporting More Complex LLM Pipelines. Our study focuses on single-prompt LLM pipelines, but more complex pipelines, such as those involving multiple prompts, external resources, and human interaction, present opportunities for auto-generating assertions for each pipeline component. For instance, in retrieval-augmented generation pipelines [28], assertions could be applied to the retrieved context before LLM responses are even generated.

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 [57] and technical users [34, 47], prompt engineering is hard for several reasons: small changes in prompt phrasing [4, 29] or the order of instructions or contexts [30] 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 [3, 11, 54, 55, 59]. Moreover, *deployed* prompts introduce new challenges, like "balancing more context with fewer tokens" and "wrangling prompt output" to meet user-defined criteria [35]. Our work doesn't focus explicitly on helping developers create better prompts, but it could indirectly support developers in improving prompts through recommended assertions.

ML and LLM Evaluation. Evaluating and monitoring deployed ML models is known to be challenging [33, 45]. Evaluating LLMs in a deployed setting is even more challenging because LLMs are typically used for generative tasks, where the outputs are free-form [13]. Some LLM pipeline types, like question-answering with retrieval-augmented generation pipelines [28], can use standardized, automated metrics [14, 39], but others face challenges due to unknown metrics and lack of labeled datasets [8, 35, 56]. Typically, organizations rely on human evaluators for LLM outputs [19, 35, 52], but recent studies suggest LLMs can self-evaluate effectively with detailed "scorecards" [7, 26, 58]. However, writing these scorecards can be challenging [35], motivating auto-generated evaluators.

LLMs for Software Testing. LLMs are increasingly being used in software testing, mainly for generating unit tests and test cases [27, 40, 48, 50, 51]. Research explores how LLMs' prompting strategies, hallucinations, and nondeterminism affect code or test accuracy [10, 15, 16, 32]. Our work is complementary and leverages LLMs to generate code-based assertions for LLM pipelines.

Testing 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, 42, 43] or provenance [31, 41]. Platforms for ML testing typically offer automated experiment tracking and prevention against overfitting [1, 38], as well as tools for data distribution debugging [20]. 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 [35]—motivating our assertion filtering approach.

6 CONCLUSION

Our work introduced a new problem of auto-generating assertions to catch failures in LLM pipelines, as well as a solution that comprises of two components: first, it synthesizes candidate assertions; then, it filters them. We proposed a taxonomy of prompt edits for assertion synthesis, demonstrating its potential via integration and deployment with LangChain. We expressed the selection of an optimal set of assertions for covering failures with high accuracy as an Integer Linear Program (ILP). We proposed assertion subsumption to cover failures in data-scarce scenarios and incorporated this into our ILP. 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.

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, Bochao 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] Anastasios N Angelopoulos, Stephen Bates, Clara Fannjiang, Michael I Jordan, and Tijana Zrnica. 2023. Prediction-powered inference. *arXiv preprint arXiv:2301.09633* (2023).
- [3] 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).
- [4] Simran Arora, Avaniika 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).
- [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] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).
- [13] 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).
- [14] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. 2023. RAGAS: Automated Evaluation of Retrieval Augmented Generation. *arXiv preprint arXiv:2309.15217* (2023).
- [15] 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).
- [16] 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).
- [17] 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>
- [18] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. In *Emerging theory, methods, and applications*. INFORMS, 257–277.
- [19] 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.
- [20] 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.
- [21] Madeleine Grunze-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).
- [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 Better Hallucinate. *arXiv preprint arXiv:2311.14648* (2023).
- [24] Daniel Kang, Nikos Arechiga, 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. EvalLM: Interactive Evaluation of Large Language Model Prompts on User-Defined Criteria. *arXiv preprint arXiv:2309.13633* (2023).
- [27] Caroline Lemieux, Jeevana Priya Inala, Shuven 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)*.
- [28] 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.
- [29] 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.
- [30] 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).
- [31] Mohammad Hossein Namaki, Avriella Floratos, 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>
- [32] 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).
- [33] 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.
- [34] 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).
- [35] 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]*
- [36] 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.
 - [37] 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).
 - [38] Cedric Renggli, Frances Ann Hubis, Bojan Karlaš, 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>
 - [39] 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).
 - [40] 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).
 - [41] 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>
 - [42] 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>
 - [43] 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.
 - [44] Shreya Shankar, Rolando Garcia, Joseph M Hellerstein, and Aditya G Parameswaran. 2022. Operationalizing machine learning: An interview study. *arXiv preprint arXiv:2209.09125* (2022).
 - [45] Shreya Shankar, Bernease Herman, and Aditya G Parameswaran. 2022. Rethinking streaming machine learning evaluation. *arXiv preprint arXiv:2205.11473* (2022).
 - [46] Shreya Shankar, Haotian Li, Will Fu-Hinthorn, Harrison Chase, J.D. Zambrescu-Pereira, Yiming Lin, Sam Noyes, Eugene Wu, and Aditya Parameswaran. 2023. SPADÉ: Automatically digging up evals based on prompt refinements. <https://blog.langchain.dev/spade-automatically-digging-up-evals-based-on-prompt-refinements/>
 - [47] 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).
 - [48] 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).
 - [49] 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).
 - [50] 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).
 - [51] 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).
 - [52] 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).
 - [53] 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.
 - [54] 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.
 - [55] 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.
 - [56] 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).
 - [57] JD Zambrescu-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.
 - [58] 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).
 - [59] 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).

A NOTATION

Table 7 summarizes the important notation used in the paper, including high-level descriptions of the ILP variables.

Symbol	Description
\mathcal{P}	A prompt template (i.e., a string with placeholders, intended to be submitted to an LLM)
\mathcal{P}_i	The i th version of a prompt template
$\Delta\mathcal{P}_i$	The diff or changes between consecutive versions of a prompt template
E	The set of all hypothetical example runs for an LLM pipeline
E'	A small set of example runs for an LLM pipeline (with labels denoting whether responses were good) that we can observe, where $E' \subset E$
n	The number of examples in E'
e_i	The i th example in E'
f	An arbitrary assertion function
F	The set of all candidate assertions from Section 2
m	The number of assertions in F
F'	A subset of F selected as a minimal set of assertions
G	Set of functions in $F \setminus F'$ not subsumed by F' (as defined by Definition 3.3)
\hat{y}_i	Binary indicator of whether example e_i satisfies all assertions in F'
y_i	Binary indicator of whether a developer considers the LLM pipeline response for example e_i good (i.e., success) or bad (i.e., failure)
α	Threshold for coverage of failing examples; F' should cover at least α of failing examples
τ	Threshold for False Failure Rate (FFR); F' should fail <i>no more than</i> τ good examples
M	$n \times m$ matrix representing the results of assertions on examples in E'
K	$m \times m$ matrix representing subsumption relationships between functions
x_j	ILP variable indicating inclusion of f_j in F'
w_{ij}	ILP variable representing if F' denotes e_i as a failure
u_i	ILP variable indicating if a failed example is covered
z_i	ILP variable defining if F' incorrectly marks a successful example as a failure (false failure)
r_j	ILP variable indicating if a function is subsumed by any function in F'
s_j	ILP variable representing functions in $F \setminus F'$ not subsumed by F'

Table 7: Notation used in the paper

B LANGCHAIN PROMPT HUB AND TAXONOMY OF PROMPT DELTAS

The LangChain Prompt Hub is an open-source repository of prompts for chains, detailing the version history of the prompts. In analyzing prompt deltas, we filter the Prompt Hub for user-uploaded chains with at least 3 prompt versions. Table 6 summarizes the chains analyzed.

C SPADE PROMPTS

SPADE leverages LLMs in three places. The first uses GPT-4 to categorize the delta (constructed by Python’s `difflib`). The second uses GPT-4 to generate Python assertion functions, based on the prompt and categories identified by the first step. The third usage of LLMs is in asking GPT-4 to evaluate whether two functions subsume each other.

Given `prompt_diff`, a list of sentences that have been modified from the previous prompt template to the current prompt template (i.e., $\Delta\mathcal{P}$), the prompt for categorizing the delta is as follows:

```
"""
Here are the changed lines in my prompt template:

`{prompt_diff}`

I want to write assertions for my LLM pipeline to run on all pipeline
responses. Here are some categories of assertion concepts I want to
check for:

- Presentation Format: Is there a specific format for the response, like a
  comma-separated list or a JSON object?
- Example Demonstration: Does the prompt template include any examples of
  good responses that demonstrate any specific headers, keys, or
  structures?
- Workflow Description: Does the prompt template include any descriptions of
  the workflow that the LLM should follow, indicating possible assertion
  concepts?
- Count: Are there any instructions regarding the number of items of a
  certain type in the response, such as "at least", "at most", or an
  exact number?
- Inclusion: Are there keywords that every LLM response should include?
- Exclusion: Are there keywords that every LLM response should never mention
  ?
- Qualitative Assessment: Are there qualitative criteria for assessing good
  responses, including specific requirements for length, tone, or style?
- Other: Based on the prompt template, are there any other concepts to check
  in assertions that are not covered by the above categories?

Give me a list of concepts to check for in LLM responses. Each item in the
list should contain a string description of a concept to check for, its
corresponding category, and the source, or phrase in the prompt template
that triggered the concept. For example, if the prompt template is "I
am a still-life artist. Give me a bulleted list of colors that I can use
to paint <object>.", then a concept might be "The response should
include a bulleted list of colors." with category "Presentation Format"
and source "Give me a bulleted list of colors".

Your answer should be a JSON list of objects within ```json``` markers,
where each object has the following fields: "concept", "category", and "
source". This list should contain as many assertion concepts as you can
think of, as long as they are specific and reasonable.
"""
```

Let concepts be the parsed categories identified by the previous prompt. The prompt for generating the Python assertion functions is as follows:

```
"""
Here is my prompt template:

`{prompt_template}`

Here is an example and its corresponding LLM response:

Example: {sample_example}
LLM Response: {sample_response}

Here are the concepts I want to check for in LLM responses:

{concepts}

Give me a list of assertions as Python functions that can be used to check
for these concepts in LLM responses. Assertion functions should not be
decomposed into helper functions. Assertion functions can leverage the
external function 'ask_llm' if the concept is too hard to evaluate with
Python code alone (e.g., qualitative criteria). The 'ask_llm' function
accepts formatted_prompt, response, and question arguments and submits
this context to an expert LLM, which returns True or False based on the
context. Since 'ask_llm' calls can be expensive, you can batch similar
concepts that require LLMs to evaluate into a single assertion function,
but do not cover more than two concepts with a function. For concepts
that are ambiguous to evaluate, you should write multiple different
assertion functions (e.g., different 'ask_llm' prompts) for the same
concept(s).

Each function should take in 3 args: an example (dict with string keys),
prompt formatted on that example (string), and LLM response (string).
Each function should return a boolean indicating whether the response
satisfies the concept(s) covered by the function. Here is a sample
assertion function for an LLM pipeline that generates summaries:

```python
def assert_simple_and_coherent_narrative(example: dict, prompt: str,
response: str):
 # Check that the summary form a simple, coherent narrative telling a
 complete story.

 question = "Does the summary form a simple, coherent narrative telling a
complete story?"
 return ask_llm(prompt, response, question)
```

Your assertion functions should be distinctly and descriptively named, and
they should include a docstring describing what the function is checking
for.
"""
```

| Task Domain | Summary of Prompt | Num. Versions |
|------------------------|--|---------------|
| Conversational AI | Act as an AI assistant that can execute tools and have a natural conversation with a user. | 8 |
| Web Development | Generate Tailwind CSS components like text, tables, and cards to help answer fantasy football questions. | 3 |
| Question Answering | Identify key assumptions in questions and generate follow-up questions to fact check those assumptions. | 5 |
| Programming Assistant | Safely execute any code a user provides to help them complete tasks, while alerting them to any concerning instructions. | 3 |
| Model Evaluation | Evaluate a model's outputs by assigning a score based on provided criteria and examples. | 6 |
| Question Answering | Concisely answer questions using no more than 3 sentences and provided context passages. | 9 |
| Workflow Automation | Create a JSON workflow using a list of provided tools based on the user's natural language query. | 11 |
| Question Answering | Answer open-ended questions by asking clarifying follow-up questions before providing a final answer. | 8 |
| Information Retrieval | Determine if a passage contains enough useful information to help answer a specific question. | 3 |
| Code Translation | Convert Python code snippets to valid, idiomatic TypeScript code. | 6 |
| Code Review | Review GitHub pull requests and provide constructive feedback for improvement. | 8 |
| Question Answering | Concisely answer questions using no more than 3 sentences and provided context passages. | 5 |
| Email Marketing | Craft a user onboarding email following marketing best practices based on provided context. | 3 |
| Text Summarization | Summarize long text into a compelling, engaging Twitter thread for a target audience. | 4 |
| Email Marketing | Craft a user onboarding email following marketing best practices based on provided context. | 7 |
| Procurement Automation | Develop a detailed, tailored negotiation strategy report using provided information about suppliers, goals, etc. | 8 |
| Education | Teach statistics topics interactively by answering questions, providing feedback, and posing example problems. | 3 |
| Fitness | Convert a text description of a fitness challenge into a structured exercise program. | 3 |
| Education | Generate engaging, concise quiz questions based on the information contained in a provided context document. | 5 |

Table 6: Description of each chain in our dataset. We describe the domain of the task the chain is trying to perform, and a short summary of the task.

C.1 Evaluating Subsumption

To evaluate subsumption, we use one prompt to query the subsumed pairs, given all assertions, and then a second prompt to format the subsumed pairs as a JSON so it can be easily parsed by SPADE. The first prompt is as follows:

```
"""
Here are all the functions I have:\n\n{assertion_blob}\n\nBased on the code,
⬇ please identify every pair of functions where one function implies the
⬇ other. Note that function A might imply function B, but function B may
⬇ not imply function A. If two functions A and B check for the same thing,
⬇ then they both imply each other (i.e., A implies B and B implies A), so
⬇ you should list both directions. Feel free to use the function names to
⬇ decide if two functions check for the same thing.
"""
```

In the prompt above, the `assertion_blob` represents a string of all assertion functions. Then, the second prompt is as follows:

```
"""
Please return your answer as a JSON list within ```json``` ticks, where
⬇ each element of the list is a tuple (A, B). If two functions A and B
⬇ check for the same thing, make sure to include both tuples (A, B) and (B
⬇ , A). For example, if I only had two functions `check_json` and `
⬇ assert_json`, the answer should be: ```json\n\n[("check_json", "
⬇ assert_json"), ("assert_json", "check_json")]```
"""
```