PromptPex: Automatic Test Generation for Language Model Prompts

RESHABH K SHARMA*, University of Washington, USA
JONATHAN DE HALLEUX, Microsoft Research, USA
SHRADDHA BARKE, Microsoft Research, USA
BENJAMIN ZORN, Microsoft Research, USA

Large language models (LLMs) are being used in many applications and prompts for these models are integrated into software applications as code-like artifacts. These prompts behave much like traditional software in that they take inputs, generate outputs, and perform some specific function. However, prompts differ from traditional code in many ways and require new approaches to ensure that they are robust. For example, unlike traditional software the output of a prompt depends on the AI model that interprets it. Also, while natural language prompts are easy to modify, the impact of updates is harder to predict. New approaches to testing, debugging, and modifying prompts with respect to the model running them are required.

To address some of these issues, we developed *PromptPex*, an LLM-based tool to automatically generate and evaluate unit tests for a given prompt. *PromptPex* extracts input and output specifications from a prompt and uses them to generate diverse, targeted, and valid unit tests. These tests are instrumental in identifying regressions when a prompt is changed and also serve as a tool to understand how prompts are interpreted by different models. We use *PromptPex* to generate tests for eight benchmark prompts and evaluate the quality of the generated tests by seeing if they can cause each of four diverse models to produce invalid output. *PromptPex* consistently creates tests that result in more invalid model outputs than a carefully constructed baseline LLM-based test generator. Furthermore, by extracting concrete specifications from the input prompt, *PromptPex* allows prompt writers to clearly understand and test specific aspects of their prompts. The source code of *PromptPex* is available at https://github.com/microsoft/promptpex.

1 INTRODUCTION

1.1 Motivation

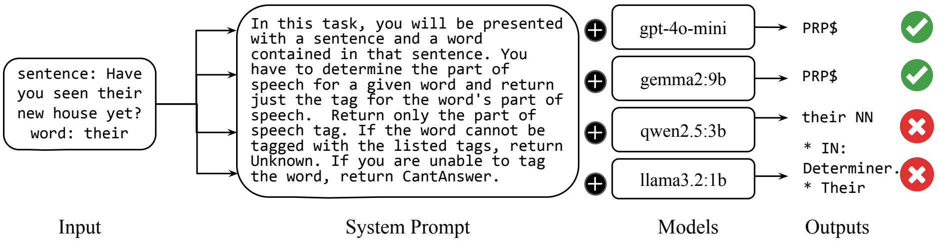
Large language models (LLMs) are being used in many applications beyond chatbots and prompts for these models are integrated into software applications as code-like artifacts. These prompts behave much like traditional software in that they take inputs, generate outputs, and perform some specific function [31]. They are also often part of complex chain of control flow that combines LLM-driven prompts and regular code supported by popular frameworks such as langchain [7].

Such prompts will become an integral part of many code bases in the future because they leverage the power of AI models and can perform common tasks such as summarization, classification, and evaluation that traditional non-AI software is unable to do. Furthermore, as AI models continue to diversify and become more efficient and effective, software that leverages them will benefit.

While prompts are becoming a key element of software code bases, they have both similarities and differences from traditional software. Similarities include taking input, generating output, performing a transformation much like an ordinary function. But differences exist that create major new software engineering challenges. First, the output of a prompt is inherently non-deterministic due to the nature of the underlying AI model and inference engine that interprets it. Significant effort has been invested in ensuring the model output conforms, at least syntactically, to a given

[*] Work done while at Microsoft Research.

Authors' addresses: Reshabh K Sharma, University of Washington, Seattle, Washington, USA, reshabh@cs.washington.edu; Jonathan de Halleux, Microsoft Research, Seattle, Washington, USA, jhalleux@microsoft.com; Shraddha Barke, Microsoft Research, Seattle, Washington, USA, sbarke@microsoft.com; Benjamin Zorn, Microsoft Research, Seattle, Washington, USA, Ben.Zorn@microsoft.com.


```
sentence: Have
you seen their
new house yet?
word: their
```

```
In this task, you will be presented
with a sentence and a word
contained in that sentence. You
have to determine the part of
speech for a given word and return
just the tag for the word's part of
speech.  Return only the part of
speech tag. If the word cannot be
tagged with the listed tags, return
Unknown. If you are unable to tag
the word, return CantAnswer.
```

```
PRP$
PRP$
their NN
* IN:
Determiner.
* Their
```

Fig. 1. Example illustrating that the use of *PromptPex* to test a given prompt against different models. For a given prompt (labeled System Prompt) and *PromptPex*-generated test input (on the left), the resulting output differs significantly depending on what AI model is used to interpret it. *PromptPex* automatically generates test input based on the prompt and evaluates whether the output is compliant with what the prompt specifies. Because the prompt specifies "Return only the part of speech tag" the lower two models produced non-compliant output.

specification [37]. Also, while natural language prompts are easy for non-programmers to write, the effect of small changes to a prompt are unpredictable, leading to challenges making robust prompt edits.

Second, unlike traditional software, where the behavior of a function depends only on the well-defined specification of the hardware that it is compiled to, the output of a prompt depends on the model that interprets it. As a result, if the AI model changes, the result of the prompt may change, sometimes dramatically [29, 32]. Application developers have strong motivations to change the underlying AI model their application uses because new models are being developed and released that are more efficient, more capable, able to run locally, available as open source, etc. As a result, it is critical that an application developer can quickly understand the implication of changing the underlying model on the behavior of a specific prompt. (Figure 1 illustrates a concrete example of this).

1.2 Our Solution: *PromptPex*

While many new software engineering practices will need to be adapted to build robust AI software[1], in this paper we focus on helping developers test and evaluate individual prompts. Specifically our tool, *PromptPex*, takes a prompt as input and automatically generates and evaluates test cases that explore whether a given model interprets the prompt as directed. *PromptPex* was inspired by work on Program Exploration (Pex [51]), now shipping as IntelliTest in Microsoft Visual Studio [35][2].

PromptPex uses an LLM to extract explicit specifications from an input prompt (the *Prompt Under Test, or PUT*) that capture the user intent in simple and concrete terms. A key element of our approach is to extract a projection of the prompt as a set of independent, concrete, checkable *output rules (OR)* that are then used to create targeted tests. Many prompts used in commercial applications have natural language statements that express such rules. For example, a prompt might contain phrases like "Ensure that..." or "The output must ..." that translate directly into our output rules.

From this extracted specification, we use an LLM to generate test cases focused on exploring whether the PUT with a given model (*the Model Under Test, or MUT*) adheres to the specification. We then run our test cases with different MUTs to generate model-specific outputs for a given test case, as shown in Figure 1. Because the tests were generated with the specification in mind, we can then

[1] In this paper, we refer to AI software as any software that uses a generative AI model at runtime.

[2] Pex automatically generates unit tests for .Net applications using dynamic symbolic execution.

automatically evaluate whether a given output from a MUT complies with the specification. While *PromptPex* cannot be used to automatically generate test cases that explore the full functionality of a given PUT, it is still valuable in creating test cases the break the requirements of our extracted specification. Details of how we extract the specifications and generate tests are provided in Section 3.

Because it is fully automated, *PromptPex* is both easy to use and provides immediate insights into potential issues that can arise from the language in the PUT, the MUT, or the combination of the two. To evaluate our approach, we collected a suite of eight benchmark PUTs, created tests using both *PromptPex* and a baseline LLM-based test generator, ran each test with four MUTs, and evaluated the tests using an LLM to determine if the outputs were compliant with requirements specified in the PUT. We consider the generation of tests that are non-compliant as more successful because a test that causes a model to generate a non-compliant output indicates a problem that should be addressed. Our results show that *PromptPex* consistently generates more non-compliant tests than our baseline test generator and also clearly distinguishes relative model capabilities for the given prompt. A full explanation of our experiments and methods is available in Section 4.

1.3 Contributions

Our contributions are as follows:

- **Systematic and Automated Test Generation for AI Model Prompts:** While prior work has focused on optimizing a prompt for a given model (e.g., [39]) and generating unit test cases for programs [25, 36, 56, 64], our work is the first to focus on the specific problem of automated test generation for prompts and creating tests that allow a developer to understand the behavior of their prompt across multiple models.
- **Specification Extraction from Prompts for Testing:** To generate effective tests, we define a new approach to extract an input specification and output rules that capture targeted properties of prompts. These generated artifacts can be used both in automatically generating tests and in helping the prompt developer test, refine, and migrate their prompts to new models.
- **Evaluation of Test Generation and Model Compliance with Specifications:** We measure the effectiveness of *PromptPex* using a benchmark suite of eight benchmark prompts running the generated tests on four diverse AI models. We compare with a sophisticated baseline LLM-based test generator and show that our approach extracting a specification from the input prompt results in tests that are more likely to cause non-compliant outputs across all the models tested.

2 MOTIVATING EXAMPLE

Chatbots, for example for customer support, are a major application of LLMs where natural language prompts describe the bot's behavior. These descriptions, or meta/system prompts, specify the role and other properties of the chatbot. They limit the domain of input and restrict the possible outputs that can be generated. Using prompts to configure a chatbot makes customization and modification easy, contributing to their widespread success. These prompts are designed to directly interact with user input, engage in conversations, and provide information to users.

The prompts we focus on, which are used as software artifacts, share some similarities but also have distinct features from chatbot prompts. First, prompts used in codebases act like programs, often with well-defined inputs and outputs. This may include both syntactic and semantic restrictions. While LLMs are generally good at handling inputs that do not adhere to format, the output must still be well-formed for other parts of the codebase, which may be traditional, to handle it effectively.

These prompts exhibit constructs similar to various programming constructs, such as performing an early return for particular input, similar to an if-then-return in traditional programming. They can also include features like complex control flow, multiple returns, assertions, and constraints. All these features are described in natural language and can be easily modified or updated.

Unlike in chatbot applications, where users can often provide any input and the chatbot needs to handle it, the input domain of a prompt embedded inside a codebase is more narrowly defined. The prompts embedded in software expect well-defined input. The constraint over the input can be explicitly stated within the prompt itself to filter out invalid input or be implicit in the codebase so that invalid input is never sent to the prompt. These prompts do not operate in isolation. They are part of complex logic pipelines where the output from a prompt or a traditional program can be fed into one another. These pipelines can be thoroughly layered, made up of multiple prompts and programs processing data to create a single response. The input to a prompt may be preprocessed to optimize efficiency or to filter out invalid data. Likewise, the output can be validated, and if necessary, the prompt can be executed again with the same input for generating a well-formed response that can be correctly parsed by dependent components [14, 15]. The main advantage of using prompts embedded in software is their capability to perform tasks on natural language inputs, which is not possible with existing traditional programs.

The prompts that are part of codebases are rich in program-like constructs. In Appendix B.6, we describe one such prompt for extracting important entities from the text and returning them in a structured list format. Due to the complexity of that prompt, we use a simpler prompt as our running example as shown in Figure 2. Although this prompt is simple, it includes program-like constructs similar to those found in complex prompts and is representative in highlighting the challenges faced by the prompts being used in traditional software pipelines.

2.1 Prompt Under Test (*PUT*)

The prompt in Figure 2 is the part-of-speech classification (POS) prompt[3], which classifies a word into a part-of-speech tag. We have truncated the list of speech tags with their description for brevity. The complete prompt is available in Appendix B.1.

> In this task, you will be presented with a sentence and a word contained in that sentence. You have to determine the part of speech for a given word and return just the tag for the word's part of speech. Return only the part of speech tag. If the word cannot be tagged with the listed tags, return Unknown. If you are unable to tag the word, return CantAnswer. Here is the alphabetical list of part-of-speech tags used in this task: CC: Coordinating conjunction, CD: Cardinal number, ...

Fig. 2. Part-of-Speech Prompt

The POS prompt has the following program-like constructs:

- **Input:** It takes a sentence and a word as input, specifying that the word must be present in the sentence.
- **Output:** It defines the output as the part-of-speech tag, `Unknown`, or `CantAnswer`.
- **Computation:** It describes how the output should be computed, which in this case is simply tagging the word as a part-of-speech tag.

[3] A modified version of the prompt from [40].

- **Control flow:** There are multiple if-then constructs in the prompt, similar to those found in code.
- **Early return:** Like in programs where the code terminates early, returning an error code, prompts also have multiple early returns to handle corner cases.
- **Assertions and constraints:** Like in programs, various assertions and constraints can be defined directly in the prompt. However, they need not be explicitly implemented. For example, the tag must be from the list of tags provided.

These prompts function like programs, featuring well-defined input and output specifications. For instance, when given the input *quick brown fox jumps over the lazy dog; quick*, the output generated is *JJ*. This input might originate from another prompt or program, and similarly, the output can be passed on to another component. While the part-of-speech prompt may align with what the prompt developer intended, ensuring that the LLM interpreting the prompt can precisely understand the intent and constraints remains a significant challenge. Developing these prompts is demanding because they require more rigorous testing than traditional software due to the vast potential input domain, even with constraints in place. Moreover, the model's behavior can vary unpredictably based on the input, which can be neither fully understandable nor predictable. The inherent ambiguity of natural language further complicates efforts to manage these challenges effectively. LLMs often struggle to consistently follow provided instructions, but these instructions can be made more precise to accurately match the intended purpose, thereby minimizing unintended consequences. As models improve their ability to follow instructions, the alignment between the prompt developer's intent and the model's actual execution behavior will become increasingly critical.

As an example of the challenges in writing an effective prompt, the POS prompt above sometimes generates a part-of-speech tag along with a description of the reasoning steps taken to arrive at that decision. This issue sometimes occurs even with *gpt-4o*, a state-of-the-art (SOTA) model. We observed this behavior due to ambiguity in what is allowed as output. To understand the ambiguity, note that the prompt requires a part-of-speech tag (e.g., NN) and not the word describing the part of speech (e.g., noun). Specifically, the prompt says "Return only the part of speech tag." The ambiguity arises because sometimes a model interprets this rule as applying only to tag (interpreting the prompt as "Return only the part of speech tag and not the word describing the part of speech") and does not explicitly forbid also adding an explanation about the reasoning behind the choice. This demonstrates the intricacies of prompt development and the value of both being more explicit about what is expected as well as testing model behavior more rigorously.

The same prompt on *gpt-4o* most of the times only outputs the tag, but on *gpt-3.5-turbo* it often prefixes the tag with *Output:*, illustrating the problem of model portability. A prompt that works correctly on one model may not function as expected on another, highlighting the importance of extensive testing across multiple models. These challenges underscore the necessity for thorough testing of prompts on various models to identify issues early and to gain insights into how prompts behave during execution on different models. This understanding can help prompt developers address any problems and optimize their prompts for improved performance across different platforms.

We developed *PromptPex*, a tool designed to help prompt developers better understand the execution behavior of their prompts. It achieves this by first extracting input and output specifications for the prompt under test (*PUT*), which are assertion-like constraints over the input and output, equivalent to pre- and post-conditions in a program.

PromptPex utilizes *gpt-4o* to extract these specifications from the prompt. The prompt developer can then examine these extracted specifications to compare their understanding with what a SOTA model perceives as expected input and the constraints on the output.

2.2 Input Specification (*IS*)

For the POS prompt, Figure 3 shows the extracted input specification (*IS*) by *gpt-4o*. For the *PUT*, the *IS* clearly captures that the input must be a sentence and a word, and it also lists constraints such as the requirement for the word to be a single word from the sentence. The single word constraint was not explicitly mentioned in the *PUT*, but during execution, the model must select a behavior, either allowing or disallowing compound words, such as *ice cream*. The *IS* highlights this under-specification in the *PUT*, indicating that the model has assumed it is expecting single words. Compound words might be considered valid input based on the developer's understanding, but they would result in undefined behavior for the model and are more likely to lead to the output `Unknown` or `CantAnswer`.

○ The input consists of a sentence combined with a specific word from that sentence.
○ The sentence must contain natural language text.
○ The word must be a single word from the provided sentence.

Fig. 3. Extracted Input Specification for Part-of-Speech Prompt

2.3 Output Specification or Rules (*OR*)

The output specification, or the rules governing the output (*OR*), captures the constraints on the output generated for *PUT*. Figure 4 shows the extracted output specification (*OR*) for the POS prompt. The prompt developer can compare these rules with their understanding. For instance, the listed rules specify that the output must consist solely of the tag, without any additional text or formatting. This can be interpreted to mean that extraneous text, such as descriptions of the tag or formatting details, is not allowed—such as the use of *Output:* seen with *gpt-3.5-turbo* but it does not restrict description of the reasoning behind the tag. These rules help the prompt developer understand the potential behavior of the prompt during execution with the state-of-the-art model. This information can be used to tune the *PUT* to generate specifications that match more accurately the intentions of the prompt developer.

○ The output must return only the part of speech tag without any additional text or formatting.
○ If the given word can be identified with one of the listed part of speech tags, the output must include only the specific tag for that word from the provided alphabetical list.
○ If the given word cannot be tagged with any of the listed part of speech tags, the output should be the word "Unknown".
○ If tagging the given word is not possible for any reason, the output should be the word "CantAnswer".

Fig. 4. Extracted Output Rules for Part-of-Speech Prompt

After first generating the *IS* and *OR*, *PromptPex* can then generate unit tests for the *PUT*. The prompt developer also has the option to edit the extracted specification to add implicit rules that are part of the pipeline in which the prompt is embedded but are not needed within the prompt itself, as the input is preprocessed. For example, the prompt developer can extend the input specification to provide the format for the input, such as *"sentence;word"*.

2.4 Test Generation

Thus far, the prompt developer is able to align the intent with the input and output specifications extracted by *PromptPex* for the SOTA model. However, this does not account for how the prompt will actually perform on various inputs, especially when tested on other models. The test inputs generated by *PromptPex* gives the prompt developer a test suite designed to cover all the output constraints in the prompt and also to challenge the model to violate the constraints set within the prompt. By running these tests, the prompt developer can identify additional failures. Figure 5 shows a few tests generated by *PromptPex* for the POS prompt. This test suite also serves as a regression test suite for any future modifications to the prompt. With passing tests, the suite now encodes the developer's intentions as concrete tests which were initially present only as specifications.

- ○ An aura of mystery surrounded them; aura
- ○ The researchers documented carefully; carefully
- ○ This is such a unique perspective; such

Fig. 5. Tests Generated for Part-of-Speech Prompt

These tests can now be executed across different models to compare and analyze the execution behavior of the prompt on other platforms. *PromptPex* assists the prompt developer in better understanding the behavior of the prompt by explicitly aligning the developer's intentions with the extracted specification and implicitly through the generated test suite. This test suite is then used to identify differences between models and determine what modifications to the original prompt are needed to achieve the intended execution across various models as envisioned by the prompt developer.

2.5 Test Evaluation

One use case for *PromptPex* is to generate tests and then allow the user to add those tests to their existing test suite. Note that *PromptPex* does not generate the correct output for each test case, so this scenario requires the user to add the correct output for the generated tests as an additional step.

PromptPex supports an automated approach to test evaluation which checks test output not for *correctness* but for *compliance with the prompt* using an LLM. We discuss our approach to test evaluation for compliance in Section 3.

3 DESIGN

In Figure 6, we present the end-to-end pipeline of *PromptPex*. It helps the prompt developer to explore prompts and understand their behavior on different models. Below, we discuss each part of the pipeline in detail.

3.1 Input Specification

Similar to *OR*, the input specification (*IS*) describes the input and the constraints expected by the *PUT*. The *IS* defines what constitutes valid input. For the POS prompt, the *IS*, shown in Figure 3,

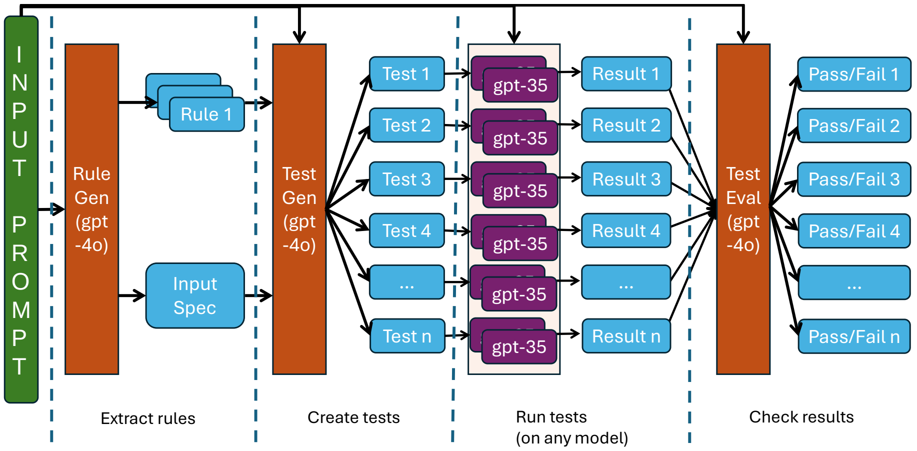


Fig. 6. The end-to-end pipeline of *PromptPex*, which is implemented using a series of LLM prompts (shown in brown). The user provides their prompt (in green on the left), and *PromptPex* automatically generates input and output specifications, and tests (in blue), and can run the generated tests on multiple models (shown in purple).

describes the input as a sentence and a word, with the constraint that the word must be present in the sentence. Some prompts may also accept input in the form of a file; in such cases, the *IS* treats file input as string content and describes the content itself as the input rather than the file. *PromptPex* also generates the *IS* using *gpt-4o*. We frame it as a task to extract the *IS* to create valid inputs. We restrict any details about the output or how the input will be used in the computation inside the *PUT*. In creating the *IS*, we first extract a description of what the inputs are. If they are composed of multiple components, those components should be listed, and their constraints and properties should be described. The complete prompt for extracting the *IS* is available in Appendix A.2. Sometimes the *PUT* will attempt to handle corner cases. For example, in the POS prompt, if the word cannot be tagged with the listed tags, it returns `Unknown`. In this case, the *IS* extractor might incorrectly assume that words which cannot be tagged are not valid inputs. While this may be true, the prompt does not imply that. We explicitly added this case in the *IS* extractor prompt to consider such inputs as part of the input domain, even if there is a rule against them.

Both the *OR* and the *IS* are editable by the prompt developer, allowing them to modify the extracted specifications or augment them with constraints from the environment.

3.2 Output Specification or Rules

The output specification or output rules (*OR*) describe the constraints on the output generated for *PUT*. Rules are concrete, checkable, general, input-agnostic, and independent constraints over the output described in natural language and derived from the *PUT*. They are similar to assertions and post-conditions in a program, detailing what the output must be and must look like, without regard for how it is generated or derived. In our running example of the part-of-speech (POS) prompt, the *OR*, as shown in Figure 4, includes rules stating that the output should either be a POS tag, `CantAnswer`, or `Unknown`. It is important to note that *OR* do not capture how the output is generated; instead, it focuses solely on constraints over the generated output. This quality of *OR* helps in keeping it separate from the input, allowing evaluation of the output based solely on the *OR*, regardless of input. The groundedness of the rules within the *OR* in the prompt is a necessary condition for the *OR* itself as it implies that all rules are valid and are present in the prompt, while the exhaustiveness in accurately covering all rules from the prompt is the sufficient condition for the *OR* as no more rules are required.

We use *gpt-4o* for extraction of the *OR* from *PUT*. The complete prompt is available in Appendix A.4. We frame it as a task to extract the rules for output validation such that the inputs are not available. We enforce the following properties during the extraction of the rules:

- If an example is present in the prompt, do not generate rules specifically for that example. Generalize them so that they will applicable for other possible inputs.
- Rules must be clear, concrete and independent from each other such that they can individually be used to validate the output.
- They should not contain any information about how the output depends on the input or how the output is computed. [4]

3.3 Test Generation

Tests for *PUT* are generated by *PromptPex* using *OR* and *IS*. The *OR* is utilized to create directed tests that challenge the model to correctly adhere to each rule, while the *IS* is used to create valid test cases. The input domain specified by the *IS* is usually extensive, as these inputs are in natural language, which can be presented in multiple forms. It is also non-trivial to determine which part of the prompt a given input covers, leaving us without a notion of coverage in prompts unlike what we have in programs.

3.3.1 Exhaustiveness. Given the vast range of possible inputs, creating an exhaustive set of test cases is challenging. To achieve this, we generate tests for each rule in the *OR*. We argue that if our *OR* is exhaustive (completely covers the prompt), the tests generated for the rules in the *OR* are also exhaustive.

PromptPex not only generates tests but also associates each test with a specific rule, providing reasoning for its creation. Beyond exhaustiveness, this approach allows for various analyses of the tests, as they are directly linked to a rule. We use this approach to attempt to develop an exhaustive test suite. Other possible future uses include modular updates to the test suites, allowing the addition of new tests for new rules and the removal of old tests linked to obsolete rules.

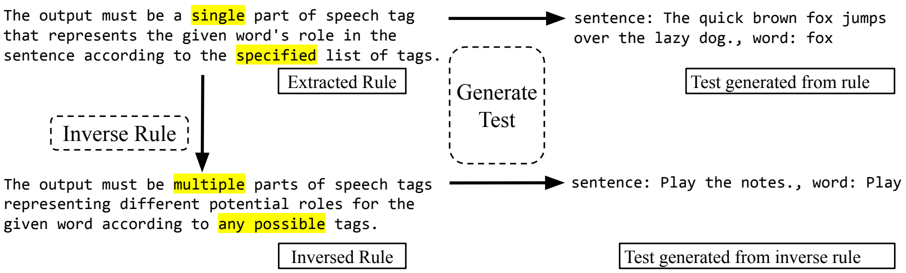
3.3.2 Generating challenging tests. Since we generate tests per rule, we can create tests that explicitly challenge these rules. We accomplish this while ensuring our test generator remains unaware of any properties of the rule itself; for a given prompt, our test generator will produce a valid test for any rule.

Inverse Rules: We generate inverse rules from the given rules. The inverse of a rule is a semantic inversion that violates the rule by describing its opposite. For example, if a rule in *OR* states that the output must always be a tag, it can be inverted into a rule that enforces the output to be the tag along with the actual name of the part of speech. *PromptPex* uses *gpt-4o* for generating inverses of the given rules. We ask it to generate inverse rules such that they contradict the given rules. The complete prompt is available in the Appendix A.5.

We generate tests for both the rules from the *OR* and their inverses as shown in Figure 7. This approach helps us generate tests that cover intriguing test cases which might cause a given model to violate the *OR*.

3.3.3 Valid Tests. A test may not follow the *IS* and might create output that violates the rules described in the *OR* and *PUT*. Such test cases are not the most valuable, as the prompt developer

[4] We limit our scope to output compliance testing where we only validate the output unlike functional testing where we validate the output for the given input.



The output must be a single part of speech tag that represents the given word's role in the sentence according to the specified list of tags.

sentence: The quick brown fox jumps over the lazy dog., word: fox

The output must be multiple parts of speech tags representing different potential roles for the given word according to any possible tags.

sentence: Play the notes., word: Play

Fig. 7. Example illustrating the use of inverse rules for test generation. Because our inverse rule was used to generate the test in the lower half of the figure, the test intentionally focuses on the word "Play" which depending on context can be labeled with multiple parts of speech.

knows that some of those inputs can be pre-filtered before they are passed to the prompt. However, these test cases might be useful for testing any input validation pipeline that processes the input or for identifying gaps in the assumed input domain. The most valuable tests are those that follow the *IS*, as they are directly actionable and require fixing. We use the extracted *IS* to guide *PromptPex* to generate tests that strictly follow the *IS*. Even in scenarios where the prompt-based system is exposed to raw inputs, the same *IS* can validate inputs and disregard any input that does not meet the input specification.

The test generator in *PromptPex* uses *gpt-4o* and takes a rule (which can also be an inverse rule), *IS* and *PUT* as input. It generates a test along with the reasoning of why the model thinks this particular test during execution will comply with the rule it was generated for. We ask the test generator to start with first understanding: What is an input? What are the different components of a valid input? What are the syntax and semantics related constraints from *IS*? In the prompt, we direct the model to consider these issues during test generation. Each test must be generated such that the expected output or behavior demonstrates adherence to the given rules for a range of scenarios, including boundary cases, typical cases, and edge cases. The exact prompt used is available in the Appendix A.5.

3.4 Test Evaluation: Compliance versus Correctness

The generated test cases can then be executed on different models. Running tests on different models helps the prompt developer understand the behavior of diverse test inputs to the prompt when run on multiple models.

As mentioned, because we lack the correct output for each test input, our automated strategy for test evaluation is an LLM-based determination of test compliance. Compliance is determined by our *LLM-as-a-judge* [65] which, given the original prompt and the test output, determines whether the output complies with the requirements described in the prompt. As a result, our evaluation explores only a partial understanding of the model output and must be augmented with additional testing. However, as demonstrated in traditional software engineering, testing the assertions over the output can also be very valuable [26, 28, 50]. After the tests are run, the results of our validator can assist the prompt developer in updating the prompt to correctly handle tests with invalid results.

To implement *LLM-as-a-judge*, we use *gpt-4o* and provide it the output generated by executing a test and the prompt (*PUT*) for which the test was executed. To allow a fair comparison with a test generator that does use *PromptPex*, we ensured that no artifacts specific to *PromptPex* are used in the validation and that the test output validator is generic, applicable to any prompt and test output regardless of the input, the method used to generate the test, or the model used to run the test.

This validator is used to evaluate the outputs of tests generated by both the baseline and *PromptPex*. We enforce the following properties for this output validator:

- Keep the evaluation independent of the input, as it will not be provided.
- Must not speculate, infer, or make any assumptions during the evaluation.
- Always check for compliance, not correctness.

The complete prompt can be found in Appendix A.1. The output validator generates a boolean result (compliant/non-compliant) and explanation for the decision.

4 EVALUATION

4.1 Evaluation Goals

In this section, we evaluate how effective *PromptPex* is at generating tests that provide actionable feedback to the developers for improving the prompts. Through our experiments, we answer the following research questions:

- RQ1: Does having specifications help in generating better tests?
- RQ2: How useful are rules for generating better tests?
- RQ3: Does having an input specification and using it to generate tests create more valid tests?
- RQ4: Can automatically generated tests help determine if a model is suitable for a prompt?

4.2 Baseline

In our evaluation, we used a zero-shot LLM-based test generator as the baseline. We use *gpt-4o* to generate tests for both the baseline and *PromptPex*. We instruct the model to develop multiple test cases by inferring the functional specification and input for the given prompt. We ask it to begin with understanding possible inputs and its components, what the syntax and semantics related constraints are, and possible input scenarios. We enforce the following properties while generating the baseline tests:

- The test cases must be designed to validate whether the output properly adheres to description.
- A good test must always be a valid input meeting the requirements mentioned in the description.
- The test cases must be diverse and distinct.
- Each test case must be crafted to rigorously assess whether the output meets the stipulated behavior based on the provided prompt.
- The input scenarios used for creating the tests must be valid, realistic, and fully comply with the given description.
- Generate test cases to broadly cover a range of scenarios, including boundary cases, typical cases, and edge cases, to thoroughly evaluate the software's adherence to the description under various conditions.
- Each test case should adhere to principles of good software testing practices, emphasizing coverage, specificity, and independence.
- Focus on creating diverse test cases that effectively challenge the prompt's capabilities by critically assessing potential weaknesses in the handling of inputs by the prompt.

The complete prompt is available in Appendix C.1. We chose this as our baseline because LLMs are capable of interpreting the prompt and generating test cases for them. LLMs are already found useful in generating unit tests for traditional software [41, 45, 49, 54, 58, 62, 63]. We ensured that the prompt used to generate the baseline tests is robust and underwent multiple revisions to enhance its ability to generate effective tests. To provide a fair comparison with *PromptPex*, we explicitly covered all requirements that are implicitly enforced by *PromptPex*, such as generating valid, challenging, and diverse test cases that comprehensively cover the prompt and help identify any flaws. Although there will always be opportunities to improve the baseline prompt, the same holds for the prompts used in *PromptPex*. We have refined both approaches sufficiently so that they can be effectively compared.

4.3 Metrics

We evaluate *PromptPex* and the baseline based on their ability to generate more effective tests.

4.3.1 % non-compliance. We define effective tests as those that expose limitations in the prompt, which means that they result in more failures. We consider non-compliance with the prompt as the metric for test quality. This includes any violations of the rules and constraints described in the prompt. This approach is similar to checking all the assertions over the output generated by a traditional program.

We used *LLM-as-a-judge* to evaluate whether the output generated by the prompt violates any rules or constraints within the prompt as described in Section 3.4. We could have also developed a validator that does not use the *PUT* but instead relies on the *OR*, as they precisely represent the constraints over the output. However, since the *OR* is part of *PromptPex* and was not provided to the baseline test generator, we refrained from using it to maintain a fair and equal comparison.

We run the tests generated by *PromptPex* and the baseline on *gpt-4o-mini*, *gemma2:9b*, *qwen2.5:3b*, *llama3.2:1b*. We compare the non-compliance of the output from the generated tests. In our evaluation, we consider more test non-compliance as better since it indicates that the tests are more effective at identifying gaps in the prompt's constraints.

4.3.2 Test Validity. Valid tests follows the input specification defined in the *PUT*. Valid tests are important as they provide instant feedback about the *PUT* and needs to be fixed. We also consider the test validity as a metric for test quality of the tests generated by *PromptPex*.

We use the *IS* with an *LLM-as-a-judge* to determine if the generated test is a valid input. The input validation prompt is available in Appendix C.2. We specifically used *IS* as *PromptPex* allows editing of generated *IS* to capture the precise input specification including implicit constraints arising from preprocessed inputs.

4.3.3 Groundedness and Spec Agreement. In addition to measuring test non-compliance and test validity which are direct indicators of the quality of the tests generated, we also evaluate the generated *OR* as these are directly used to generate tests by *PromptPex* and have impact on the quality of the generated tests.

A rule in the *OR* is considered grounded if it is present in the *PUT*. We do not want to generate tests for the rules which are not present in the *PUT*. We consider a higher groundedness of the rules as an indirect metric for test quality. To determine if the generated rules are grounded in the original *PUT*, we used *LLM-as-a-judge* and ask it to confirm rule groundedness. The complete prompt for checking the groundedness of the rules is available in Appendix C.3.

Another goal for generating *OR* is to ensure that all important constraints mentioned in the *PUT* are captured in the *OR*. We refer to this as spec agreement. To compute this, we extract a description of how the output must be computed from the *PUT*, the *task specification.* The prompt for extracting the task specification is available in Appendix C.4. We append the task specification with our extracted *OR* to derive a *spec prompt* which in theory should capture all the same constraints as the

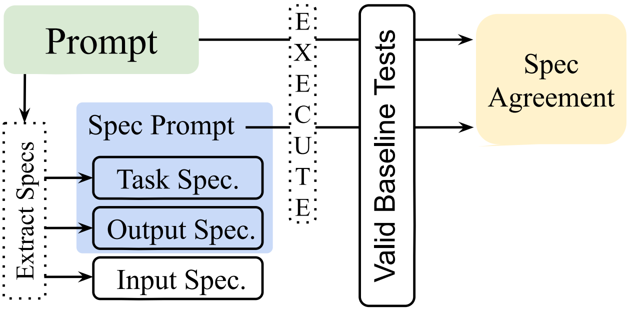


Fig. 8. Overview of the spec agreement estimation for the tests generated by *PromptPex*. The specifications are extracted from the prompt and a spec prompt is created by appending the output specification to the task specification. The valid baseline tests are then executed with the original and synthetic prompt. Spec agreement is considered high when the spec prompt and the original prompt behave similarly. This is estimated by comparing the number of non-compliant test results.

PUT. We do not use the *IS* in the spec prompt because *PromptPex* generates tests for each rule in the *OR*. To estimate how closely the extracted specification represents the original *PUT*, we compare the spec prompt with the original *PUT*. We feed tests generated from our baseline test generator to both the spec prompt and the *PUT* to compare their behaviors as shown in Figure 8. We use the baseline tests instead of the *PromptPex* tests so that they are not influenced by our process of generating them from the *OR*. The spec prompt and the original *PUT* must have similar behavior for the generated *OR* to have high spec agreement. We use cosine similarity of the non-compliance percentage of spec prompt and the original *PUT* to derive a spec agreement score.

4.4 Benchmarks

We list the prompts with sources and their descriptions in Table 1. We selected a diverse set of prompts from publicly available sources, focusing on those that are within the scope of *PromptPex*. Currently, we support prompts that can accept only a single input. Although this single input can be interpreted by the model to be made up of multiple components, this differentiation is not explicitly present, for example, a single string as input representing a sentence and a word for the speech tag prompt. This limitation makes it unsuitable for prompts requiring multiple embedded inputs. We also only support prompts where the output is independent of the previous outputs, making prompts describing multi-turn conversations for tasks out of scope for *PromptPex*.

4.5 Evaluation Procedure

We accessed *gpt-4o* and *gpt-4o-mini* through APIs. We kept the temperature 1.0 across all the requests. We used Ollama [10] for the local models, *gemma2:9b*, *qwen2.5:3b*, *llama3.2:1b*. We ran all the experiments and hosted the local models on a virtual machine (VM) hosted on Microsoft Azure. Our system runs on an AMD EPYC 7V13 processor with 256 GB of RAM, 2 TB of SSD, and an NVIDIA A100 GPU with 80 GB of dedicated memory. The VM is running Ubuntu 22.04.5 LTS and is allocated 24 out of 64 CPU cores. All the prompts are written and ran using GenAIScript [4] and Prompty [13]. We ran each test once per prompt per model.

4.6 How useful are specifications for generating tests? (RQ1)

To test our hypothesis that having explicit specifications like *OR* and *IS* helps in generating better tests, we generated tests for the different prompts in our benchmark using *PromptPex* and the

Name of Prompt	**Prompt Description**	**Source**
speech-tag	Determine the part of speech for a given word within a sentence using a predefined set of tags. The task may return tags such as noun, verb, adjective among others, or return "Unknown" or "CantAnswer" if the word cannot be classified.	Modified from an example used by Schnabel et. al. [40]
text-to-p	Format a paragraph of text into HTML by splitting it into sentences and wrapping each sentence with paragraph tags, while enhancing key words and phrases with additional HTML tags.	format_text prompt from Ghost-Writer [22]
shakespeare	Assist users in creating text that mimics the Shakespearean style of writing, including the use of archaic language and stylistic elements typical of the period.	Azure AI Studio Prompt Catalog [16]
sentence	Rewrite a sentence to improve its readability and make it more conversational while preserving its original meaning. This includes simplifying complex phrases and enhancing engagement through fluid structure.	The Big Prompt Library [17]
extract-names	Extract model names from machine learning paper abstracts, returning a structured array of identified model names or "NA" if none are found.	Information extraction prompt from Prompt Hub [21]
elements	Extract important entities from a text, including company names, people names, specific topics, and general themes, presented in a structured list format.	OpenAI documentation [20]
classify	Classify a news article into one of several predefined categories such as World, Sports, Business, or Sci/Tech, based on its content and context.	Prompt used in a tutorial [19]
art-prompt	Create detailed prompts based on user descriptions for generating AI images, focusing on key characteristics, timing, lighting, and the desired emotional impact of the image in a concise single paragraph.	The Big Prompt Library [18]

Table 1. Description of the prompts used in the evaluation with their sources.

baseline. To compare the quality of test generation, we compare the percentage of tests not compliant with the prompt description for *PromptPex* and the baseline on multiple models. Higher percentage of non-compliance is better as it represents more test failures. Table 2 shows the percentage of test non-compliance for different prompts on each model. *PromptPex* on average generated 5.5% more tests for different models that were not compliant with description of the *PUT* from the benchmark.

Figure 9 shows the average non-compliance across the benchmarks for *PromptPex* and baseline for each of the different models. The more capable models (like *gpt-4o-mini*) have lower rates of non-compliance while the smaller models (like *llama3.2:1b*) have high rates. *PromptPex* beats baseline in every case but is particularly more effective for *gpt-4o-mini*, where the baseline model has a much harder time generating any tests that result in non-compliant output.

Prompts	gpt-4o-mini		gemma2:9b		qwen2.5:3b		llama3.2:1b	
	PPex	BL	PPex	BL	PPex	BL	PPex	BL
speech-tag	0%	0%	5%	2%	2%	7%	90%	98%
text-to-p	15%	5%	23%	38%	72%	69%	95%	97%
shakespeare	2%	2%	5%	2%	5%	5%	7%	7%
sentence	12%	0%	4%	6%	21%	2%	31%	19%
extract-names	0%	0%	6%	16%	31%	22%	61%	46%
elements	31%	2%	44%	15%	62%	19%	54%	62%
classify	4%	0%	0%	0%	12%	0%	25%	25%
art-prompt	2%	0%	17%	12%	17%	7%	48%	45%
Average	8%	1%	13%	11%	28%	16%	51%	50%

Table 2. Test non-compliance results for the tests generated by *PromptPex*(PPex) vs the baseline (BL) on different models. More test non-compliance (winner shown in blue) represents more challenging tests and hence are better tests.

Prompts	gpt-4o-mini		gemma2:9b		qwen2.5:3b		llama3.2:1b	
	RL	Inv	RL	Inv	RL	Inv	RL	Inv
speech-tag	0%	0%	0%	10%	5%	0%	95%	86%
text-to-p	14%	17%	14%	33%	62%	83%	90%	100%
shakespeare	0%	5%	0%	10%	0%	10%	0%	14%
sentence	17%	8%	0%	8%	25%	17%	38%	25%
extract-names	0%	0%	0%	11%	12%	28%	56%	67%
elements	33%	29%	17%	71%	58%	67%	50%	58%
classify	0%	8%	0%	0%	0%	25%	25%	25%
art-prompt	0%	5%	5%	29%	10%	24%	29%	67%
Average	8%	9%	4%	21%	24%	32%	48%	56%

Table 3. Test non-compliance results for tests generated from rules (RL) and tests generated from inverse rules (Inv) by *PromptPex* on different models. Higher test non-compliance (winner shown in blue) represents better tests.

4.7 How useful are rules for generating better tests? (RQ2)

We saw that *PromptPex* generates better tests using its approach, but we also evaluate how important is the fact that we generate tests for specific rules and inverse rules. As discussed in Section 3.3.2, we extract *OR* from the *PUT*. The *OR* is made up of rules which are the constraints over the output. We create inverse rules from these rules and use the generated rules to create tests. This approach allows us to compare the non-compliance of tests generated from the *OR* and from the inverse *OR*. Table 3 shows the percentage of their test non-compliance for different prompts on each model. The tests generated from the inverse rules on average generated 8.5% more non-compliant tests as shown in Figure 10 demonstrating the role of the *OR* and the rules whose inverse created these tests. Notably the tests generated for the inverse rules were able to consistently generate better tests even for a large model like *gpt-4o-mini* which otherwise is challenging even for the tests directly generated from *PromptPex* rules.

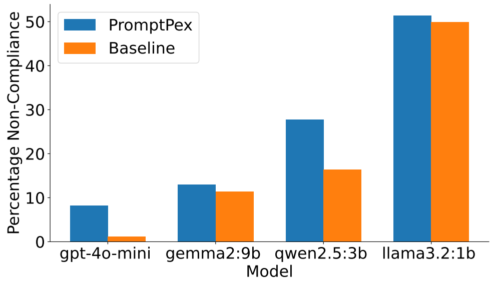


Fig. 9. Average % Test Non-Compliance of tests generated by *PromptPex* and Baseline.

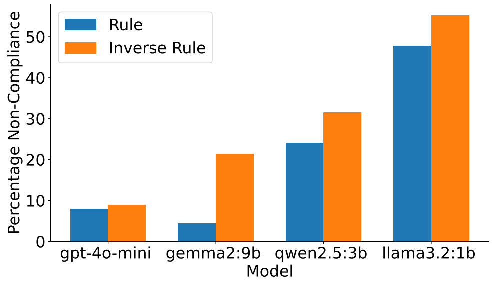


Fig. 10. Average % Test Non-Compliance of the tests generated for rules and inverse rules by *PromptPex*.

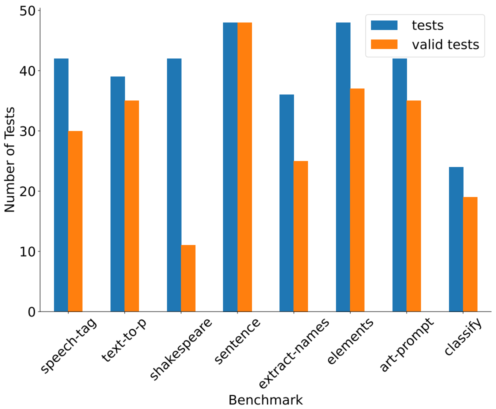


Fig. 11. Number of valid tests generated by *PromptPex*.

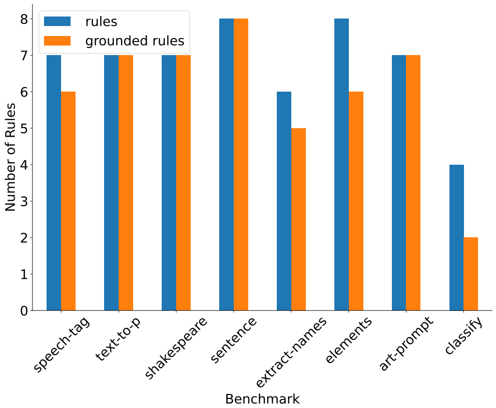


Fig. 12. Number of grounded rules generated by *PromptPex*.

4.8 Does having an input specification help generate valid tests? (RQ3)

PromptPex uses *IS* to generate tests that meet the constraints on the input as described in *PUT*. In Figure 11, we show the total number of *PromptPex* generated tests for each benchmark as well as the number of tests considered valid by our validation analysis. We observe that a significant fraction of all generated tests are deemed valid, with the shakespeare prompt being an exception due to the input validator considering non-shakespearean input as invalid[5]. We also compare (not shown) the percentage of compliant valid tests with all the tests generated by *PromptPex*. We observed that 73% of all the non-compliant tests are valid tests. Non-complaint valid tests are valuable as they act as instant feedback and must be fixed.

4.9 Can automatically generated tests help determine if a model is suitable for a prompt? (RQ4)

Prompt developers have many AI model options to choose from for a given application. They need to understand how effective a model is for their specific prompt. Because *PromptPex* automatically

[5] Writing prompts that process other prompts is challenging. In particular, our input validator in this case was confused and thought the input should be shakespearean text as well.

generates tests, we seek to understand if the automatically generated tests can appropriately distinguish model effectiveness for a given prompt.

Figure 13 shows the *PromptPex* test % non-compliance for the models under test across the benchmarks. It illustrates that for a given prompt, tests generated by *PromptPex* strongly differentiates the capabilities of the models and can be used to determine model suitability. The more capable models (like *gpt-4o-mini*) have lower rates of non-compliance while the smaller models (like *llama3.2:1b*) have high rates. At the same time, some of the smaller models are highly capable for a given prompt. For example, both the *gemma2:9b* and *quen2.5:3b* models are effective for the speech-tag benchmark.

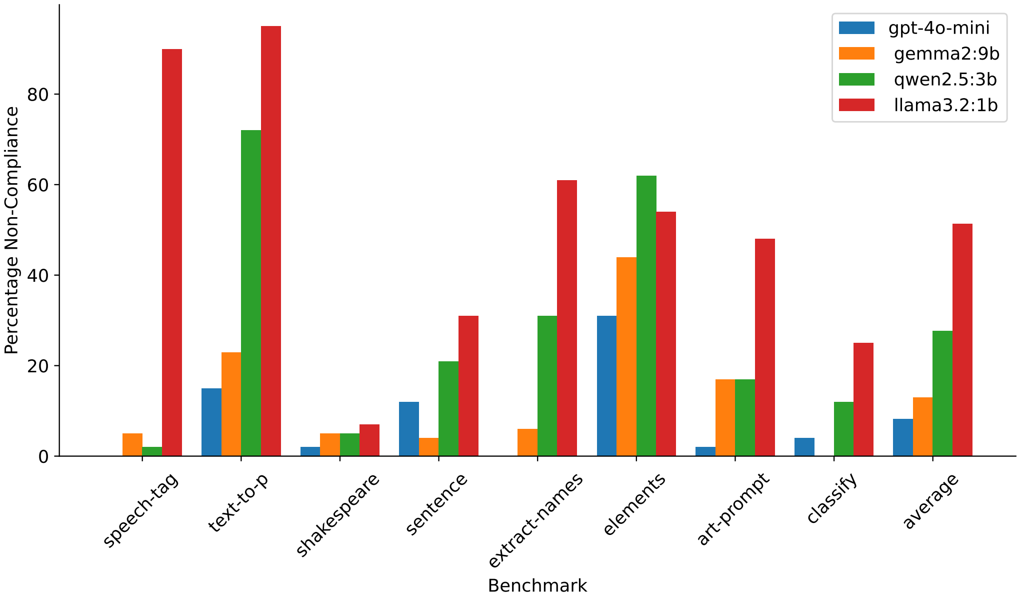


Fig. 13. % Test Non-Compliance of tests generated by *PromptPex*.

4.10 Groundedness and Spec Agreement

It is important that the generated *OR* are both grounded in the *PUT* and cover all the important constraints expressed in the *PUT*.

In Figure 12, we show how many rules were generated for each benchmark and whether those rules were grounded, according to our groundedness evaluation. We found that on an average 89% rules from *OR* were grounded in the respective prompt from the benchmark with the classify prompt being an exception. The classify prompt takes input as a news article and classifies it into a given category. The non-grounded rules are generated by the model to compensate for the under-specification in the prompt. The classify prompt does not cover the cases when the output can be multiple categories or should the output just be the name of category without any explanation.

We observe high scores for spec agreement for all the prompts in our benchmark (96.8%) with the shakespeare prompt being an exception, without it the prompts achieved the spec agreement score of 99.9%.

High groundedness and spec agreement score ensures that the *OR* generated is neither over or under extracted respectively and the tests generated by *PromptPex* are focused on testing the constraints in the *PUT*.

5 DISCUSSION

In this section, we consider concrete examples from our benchmarks to better understand how *PromptPex* can help with the development of prompts. *PromptPex* not only generates tests but also generates specifications that are used for generating tests and have value on their own.

5.1 Input Specification

IS, which represents the constraints over the input, can be used to understand what the model expects as input for a particular prompt. This can be extremely useful when the input is complex or not obvious when reading the prompt. The developer can ensure the input spec remains the same with changes to the prompt that do not intend to change the input. It can also help in understanding if the input is under-specified in the prompt.

For example, for the elements prompt (Appendix B.6), the *IS* states that the input text can be of any length and can be in any language. This means that the model is expecting some implicit understanding of the input, as these constraints were not mentioned in the prompt itself, which only stated that the input is text.

In the art prompt (Appendix B.8), the description of the input in the prompt is confusing. The prompt takes user descriptions as input and converts them into prompts for generating art. The prompt talks about the input description and the generated prompt in a similar way, which is even confusing to read. It is not straightforward to separate the specification for the input and output. For example, `ensuring each description does not exceed 80 words and is crafted in a single paragraph`, the description refers to the input user description, but "crafted" can also mean it is referring to the generated prompt, which must be in a single paragraph. Such ambiguities are also reflected in the generated input specification, where an output constraint about the generated prompt being in English is present in *IS*. The prompt writer can take a look at the *IS* and update their prompt to make it clearer and without ambiguities.

Similarly, in the classify prompt (Appendix B.7), the prompt redefines the notion of a news article by stating that a news article can be classified into the following categories, which is reflected in the *IS*. If this was the actual intent, then it is okay; otherwise, seeing the *IS* provided the useful feedback to change the way a news article is defined in the prompt to make it more generic.

IS not only helps in generating tests that are valid but also helps the prompt developer understand if the input is properly specified in the prompt and if there is some ambiguity or errors around it.

5.2 Output Specification

The same can be said about the *OR*; it can also be used to understand what the model thinks about the output that can be generated. It can be used to figure out any under-specification or errors in the output constraints defined in the prompt.

For example, consider the elements prompt (Appendix B.6), where the values for multiple labels are extracted from the input. The *OR* states a rule that the labels must still be present even when there is no value that can be extracted for them. This case is not defined in the prompt, but it is a decision the model needs to take to handle such cases, thus pointing out an under-specification.

Similarly, for the art prompt (Appendix B.8), as we saw above while observing the *IS*, the *OR* is also getting confused, and there is a rule about only generating a description of length 80 words, which must be applied to the input. A prompt developer, after seeing the *OR* and *IS*, will be able to clarify the input and output in the prompt such that they generate better and clearer specifications, which also match the intent of the developer.

For the classify prompt, the *OR* adds a rule about not outputting anything other than the name of the category, pointing towards an under-specification, i.e., does it want the name of the category with or without any explanation, etc.

From these examples, it is clear that the specifications help in revising the prompt to improve any under-specification over the input and output. They can also be used to easily identify conflicting rules, which we did not see because the prompts we used were well-formed. The specifications can also be used to highlight bugs, like the inconsistencies found in the art prompt.

5.3 Inverse Rules

When we create tests from the rules in *OR*, we generate tests aimed at ensuring they follow the rule. Such tests ensure that the output is able to comply with the rules, and if it fails, they can help expose situations when there are conflicting rules causing the failure or when the model is not capable of generating output compliant with the rules. For a fairly capable model and a well-written prompt, these tests are expected to pass.

Since inverse rules contradict the rules themselves, they attempt to push the model to generate output that does not comply with the rules. We have shown an example of a test generated from the inverse rule in Figure 7, where the generated test is created such that it can potentially confuse the model and output multiple tags for the same word. These tests also have limitations as they are bounded by the input specification, which limits how challenging they can be. For example, for the speech tag prompt (Appendix B.1), in cases when the output can be `Unknown` or `Can't Answer`, the test might contain a made-up word, a number, or an empty string. However, all of these are outside the input domain, which forces the test generator to create valid tests that are challenging.

5.4 Comparison with Baseline Tests

We examined the differences between the tests generated by *PromptPex* and those from the baseline to highlight some distinctive properties found in the tests created by *PromptPex*.

5.4.1 Creativity. Tests generated by *PromptPex* are more creative than those produced by the baseline while still remaining within the input domain. For instance, in the speech tag prompt (Appendix B.1), *PromptPex* used non-existent words in tests like `sentence: The truth was uncertain, shenative., word: shenative` and `sentence: The xylophone zxylophone harmonizes., word: zxylophone` while the most creative attempt from the baseline was the use of `12` as a word, which lies outside the input domain. For the classify prompt (Appendix B.7), tests generated by *PromptPex* could be classified into multiple categories, like `Google announces groundbreaking quantum computing progress`, which fits both tech and business categories, unlike the baseline, which did not create ambiguous tests. In the extract name prompt (Appendix B.5), *PromptPex* produced many more tests with less common or imaginary model names such as `ReinforceNet and AdvancedDL` in varying contexts than the baseline did. For the sentence rewrite prompt (Appendix B.4), *PromptPex* addressed complex subject matter, such as `the proliferation of digital technologies` and `integration of quantum computing`, while the baseline focused on routine subjects. Lastly, in the Shakespeare prompt (Appendix B.3), *PromptPex* explored modern scenarios like `Compose a modern dialogue about picking a TV show to watch` and `Write an excuse note for not doing my chores` whereas the baseline was confined to traditional themes.

5.4.2 Complexity. PromptPex also produced more complex tests compared to the baseline. In the speech tag prompt (Appendix B.1), it used more complex words, for example, `sentence: She spoke` `eloquently although about nothing in particular., word: although`. For the classify prompt (Appendix B.7), *PromptPex* generated tests with varied descriptions, unlike the consistent tests from the baseline. In the sentence rewrite prompt (Appendix B.4), *PromptPex* used complex language and demonstrated conceptual depth with tests like `juxtaposition of chaos and order in contemporary art`, while baseline tests remained more literal. In the Shakespeare prompt (Appendix B.3), *PromptPex* introduced complexity by creating scenarios involving layered themes and character interactions, examples being `Describe a character's internal conflict in a scene` and `Write a dramatic scene about a kingdom's downfall`.

6 LIMITATIONS

PromptPex, while promising, faces several key limitations. The system is currently designed to handle single-input prompts as a single string, which can be interpreted as multiple components, but lacks support for structured inputs like embedded prompts. Additionally, *PromptPex* does not support generating dummy RAG data as input alongside tests, limiting its applicability in prompts which also take RAG data as input.

Many prompts used in *PromptPex* take another prompt as input creating risk of prompt injection attacks. We filter out the malicious inputs using Azure AI Content Safety [2]. Despite following best practices and proper prompt delimitation, the potential for injection attacks can affect the specification extraction, test generation, and evaluation. We use *LLM-as-a-judge* for output validation, though the use of LLMs for output validation, while effective, is not 100% reliable [53] and may impact the system's overall accuracy. LLMs also exhibit self-bias when evaluating their own outputs [55]. *PromptPex* rely on *gpt-4o* for both test generation and test output validation risking the overall accuracy due to the propagation of these biases. We consider using code-based validation methods as a future improvement to address some of these issues.

Furthermore, *PromptPex* faces parsing challenges due to occasional formatting issues in generated results, which can lead to failures in processing artifacts. We plan to implement structured output as future work to mitigate these issues. *PromptPex* consistently generates more concise tests compared to the baseline, which, while not observed to reduce quality, may be less preferable in contexts requiring more detailed descriptions, such as abstract or note generation. These limitations collectively provide clear directions for future work and improvements to *PromptPex*.

7 RELATED WORK

Existing research has broadly highlighted the need for prompt testing [34, 38]. However, existing work has focused more on using unit test suites to assist model migration for a prompt [29] or as a regression test suite for checking future modifications to the prompt [42]. *PromptPex* automatically generates tests using the extracted specification from the prompt, and this test suite can be used for model migration or as a regression test suite.

7.1 Prompt Testing

A prompt can result in different outputs when executed on different models. Even a single word change in a prompt can lead to drastic changes in the output. This makes prompt testing important, and currently, it is mostly done manually or through the use of benchmarks to evaluate a prompt's performance on specific tasks like classification [24]. Model migration and regression also expect the user to provide a manually created unit test suite.

Several prompt development platforms offer support for writing unit tests for prompts [1, 3, 6, 8, 12]. Promptfoo [1] allows for writing assertions over generated output and SPADE [42] automatically generates similar assertions over the output for checking regression but none of these platforms automatically generate unit tests.

7.2 Prompt Fuzzing

Prompt fuzzing also addresses similar problems as *PromptPex* by generating tests to highlight flaws in the prompt. The key difference is that fuzzing also checks the prompt's ability to handle ill-formed inputs, whereas *PromptPex* focuses on creating valid inputs. Even when it generates invalid tests, prompt fuzzing is closely related to our work as it involves generating tests for prompts. Currently, the primary focus of most prompt fuzzing efforts is the creation of adversarial inputs for red teaming [30, 57, 59–61]. The automatic test generation in promptfoo [9] is also focused on generating malicious prompts. While exploring the impact of malicious inputs is valuable, *PromptPex* has a broader focus on generating inputs that help evaluate the robustness of the combination of the *PUT* and model under test.

7.3 Prompt Optimization

Prompt optimization involves creating prompt variants that perform better on a specific model or are more cost-effective by being smaller. Although prompt optimization is not directly related to our work, most techniques for it require an equivalence checking test suite. Often, this suite consists of a set of manually labeled examples, either from the user or from commonly available benchmarks [27, 33, 39, 47]. Some methods use these datasets as seed examples and generate additional examples from them [23]. The generation of examples in such cases is more comparable to the baseline we used in our evaluation than to *PromptPex*.

7.4 Synthetic data generation

There is an increasing demand for high-quality, diverse data for different phases of LLMs, which existing datasets cannot meet. Synthetic data not only fulfills this requirement but also fills gaps where specialized data is needed, for example, data with reasoning steps [48, 52]. Synthetic data generation shares a similar goal with *PromptPex* in that both generate input data for a specialized task. However, most efforts in synthetic data generation are not done from scratch, and even when they are, the methodology is more similar to the baseline used in our evaluation than to *PromptPex*.

7.5 Prompt Specification

In *PromptPex*, we extract input and output specifications to serve multiple purposes, from understanding the prompt to generating input or output validators and unit tests. In parallel, Stoica et al. [46] have also highlighted the importance of specifications in prompt engineering for making it as reliable and robust as traditional software engineering. Although we could not find any work on extracting and using specifications for generating tests, we found research on using specification for input and output validation and output generation. Sharma et al. [43] define input specifications for a VLLM prompt using a declarative meta-language, SPML [44]. Amazon Bedrock Guardrails [11] allows defining policies over the output in a natural language abstraction over formal logic using declarative variables with natural language description. Structured or constrained decoding [5] generates output that follows a given domain-specific grammar, which is equivalent to an output structure specification.

8 CONCLUSION

AI model prompts are an increasingly common and important part of many software applications yet few tools exist to help software developers write, test, debug, maintain, and migrate such prompts to new models.

We have presented *PromptPex*, the first LLM-based tool that, given an AI model prompt as input, automatically generates an input and output specification for the prompt as well as test cases to explore the behavior of the prompt. The specifications and test cases alone are valuable artifacts that can help the prompt developer understand their prompt and test it. Tests generated by *PromptPex* can be used to augment existing unit test cases for prompts if they already exist. Further, *PromptPex* automatically tests a given prompt and test input against a collection of AI models and evaluates the results for output compliance with the input prompt, allowing users to quickly understand the behavior of their prompt on multiple models.

A key element of our approach is that we extract a projection of the prompt as a set of independent, concrete, checkable output rules that are then used to create targeted tests. Many prompts used in commercial applications have natural language statements that express such rules. For example, a prompt might contain phrases like "Ensure that..." or "The output must ..." that translate directly into our output rules.

In evaluating *PromptPex* with a suite of eight prompt benchmarks using four diverse AI models, we show that *PromptPex* consistently outperforms the baseline LLM-based test generator and clearly identifies which models are most suited to use for given prompt. Future extensions to *PromptPex* will focus on test generation for more sophisticated inputs, extracting more sophisticated logical constraints expressed in prompts, and integrating our test generation with prompt optimization approaches.

REFERENCES

[1] Assertions & metrics | promptfoo — promptfoo.dev. https://www.promptfoo.dev/docs/configuration/expected-outputs/. [Accessed 13-12-2024].

[2] Azure AI Content Safety – AI Content Moderation | Microsoft Azure — azure.microsoft.com. https://azure.microsoft.com/en-us/products/ai-services/ai-content-safety. [Accessed 09-01-2025].

[3] Create Test Sets | Agenta Documentation — docs.agenta.ai. https://docs.agenta.ai/evaluation/create-test-sets. [Accessed 13-12-2024].

[4] Generative AI Scripting — microsoft.github.io. https://microsoft.github.io/genaiscript/. [Accessed 09-01-2025].

[5] GitHub - guidance-ai/guidance: A guidance language for controlling large language models. — github.com. https://github.com/guidance-ai/guidance. [Accessed 05-02-2025].

[6] GitHub - openai/evals: Evals is a framework for evaluating LLMs and LLM systems, and an open-source registry of benchmarks. — github.com. https://github.com/openai/evals. [Accessed 13-12-2024].

[7] LangChain — python.langchain.com. https://python.langchain.com. [Accessed 13-12-2024].

[8] LangChain — python.langchain.com. https://python.langchain.com/docs/concepts/testing/#unit-tests. [Accessed 13-12-2024].

[9] LLM Vulnerability Scanner | promptfoo — promptfoo.dev. https://www.promptfoo.dev/llm-vulnerability-scanner/. [Accessed 05-01-2025].

[10] Ollama — ollama.com. https://ollama.com/. [Accessed 09-01-2025].

[11] Prevent factual errors from LLM hallucinations with mathematically sound Automated Reasoning checks (preview) | Amazon Web Services — aws.amazon.com. https://aws.amazon.com/blogs/aws/prevent-factual-errors-from-llm-hallucinations-with-mathematically-sound-automated-reasoning-checks-preview/. [Accessed 14-01-2025].

[12] PromptLayer — docs.promptlayer.com. https://docs.promptlayer.com/quickstart#evaluations. [Accessed 13-12-2024].

[13] prompty.ai — prompty.ai. https://prompty.ai/. [Accessed 09-01-2025].

[14] TypeChat — microsoft.github.io. https://microsoft.github.io/TypeChat/. [Accessed 05-01-2025].

[15] Validators - Pydantic — docs.pydantic.dev. https://docs.pydantic.dev/latest/concepts/validators/. [Accessed 05-01-2025].

[16] Azure ai studio prompt catalog. https://ai.azure.com/explore/prompts/shakespeare_writing_assistant/version/0.0.1/registry/azureml?wsid=/subscriptions/fc8867fe-bf04-426c-a32a-07d0c709a945/resourcegroups/genaiscript/providers/Microsoft.MachineLearningServices/workspaces/genaiscript&tid=512451b2-ca3c-4016-b97c-10bd8c704cfc&promptType=promptSamples&promptSharedInHub=false, 2023.

[17] The big prompt library. https://github.com/0xeb/TheBigPromptLibrary/blob/main/CustomInstructions/ChatGPT/SaxlWzH4g_Sentence_Rewriter_Tool.md, 2023.

[18] The big prompt library. https://github.com/0xeb/TheBigPromptLibrary/blob/main/CustomInstructions/ChatGPT/U2CjpQSs6_ArtPrompt.md, 2023.

[19] How to use llama2 tutorial for text classification. https://pupuweb.com/how-use-llama-2-text-classification-tasks/, 2023.

[20] Openai documentation: Best practices for prompt engineering with the openai api. https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api, 2023.

[21] Prompt examples from the website. https://www.promptingguide.ai/prompts/information-extraction/extract-models, 2023.

[22] Ghostwriter: Augmenting collaborative human-ai writing experiences through personalization and agency. https://arxiv.org/abs/2402.08855, 2024.

[23] Eshaan Agarwal, Vivek Dani, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware agent-driven prompt optimization framework. *arXiv preprint arXiv:2405.18369*, 2024.

[24] BIG bench authors. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research*, 2023.

[25] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative ai: A comparative performance analysis of autogeneration tools. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 54–61, 2024.

[26] Margaret Burnett, Curtis Cook, Omkar Pendse, Gregg Rothermel, Jay Summet, and Chris Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 93–103. IEEE, 2003.

[27] Yuyan Chen, Zhihao Wen, Ge Fan, Zhengyu Chen, Wei Wu, Dayiheng Liu, Zhixu Li, Bang Liu, and Yanghua Xiao. Mapo: Boosting large language model performance with model-adaptive prompt optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, page 3279–3304. Association for Computational Linguistics, 2023.

[28] Lori A Clarke and David S Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[29] Tanay Dixit, Daniel Lee, Sally Fang, Sai Sree Harsha, Anirudh Sureshan, Akash Maharaj, and Yunyao Li. Retain: Interactive tool for regression testing guided llm migration, 2024.

[30] Xueluan Gong, Mingzhe Li, Yilin Zhang, Fengyuan Ran, Chen Chen, Yanjiao Chen, Qian Wang, and Kwok-Yan Lam. Effective and evasive fuzz testing-driven jailbreaking attacks against llms, 2024.

[31] Tommy Guy, Peli de Halleux, Reshabh K Sharma, and Ben Zorn. Prompts are programs. https://blog.sigplan.org/2024/10/22/prompts-are-programs//, 2024. SIGPLAN Perspectives Blog, [Accessed 18-12-2024].

[32] Eaman Jahani, Benjamin S. Manning, Joe Zhang, Hong-Yi TuYe, Mohammed Alsobay, Christos Nicolaides, Siddharth Suri, and David Holtz. As generative models improve, we must adapt our prompts, 2024.

[33] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[34] Wanqin Ma, Chenyang Yang, and Christian Kästner. Why is my prompt getting worse? Rethinking regression testing for evolving LLM APIs. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, CAIN 2024, page 166–171. ACM, April 2024.

[35] Microsoft. Overview of Microsoft IntelliTest. https://learn.microsoft.com/en-us/visualstudio/test/intellitest-manual/?view=vs-2022, 2024. [Accessed 18-12-2024].

[36] Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. Test oracle automation in the era of llms. *ACM Transactions on Software Engineering and Methodology*, 2024.

[37] OpenAI. OpenAI API guide: Using JSON mode. https://community.openai.com/t/openai-api-guide-using-json-mode/557265, 2024. [Accessed 18-12-2024].

[38] Chris Parnin, Gustavo Soares, Rahul Pandita, Sumit Gulwani, Jessica Rich, and Austin Z. Henley. Building your own product copilot: Challenges, opportunities, and needs, 2023.

[39] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

[40] Tobias Schnabel and Jennifer Neville. Symbolic prompt program search: A structure-aware approach to efficient compile-time prompt optimization. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 670–686, 2024.

[41] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024.

[42] Shreya Shankar, Haotian Li, Parth Asawa, Madelon Hulsebos, Yiming Lin, J. D. Zamfirescu-Pereira, Harrison Chase, Will Fu-Hinthorn, Aditya G. Parameswaran, and Eugene Wu. spade: Synthesizing data quality assertions for large language model pipelines. *Proceedings of the VLDB Endowment*, 17(12):4173–4186, August 2024.

[43] Reshabh K Sharma, Vinayak Gupta, and Dan Grossman. Defending language models against image-based prompt attacks via user-provided specifications. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 112–131. IEEE, 2024.

[44] Reshabh K Sharma, Vinayak Gupta, and Dan Grossman. Spml: A dsl for defending language models against prompt attacks. *arXiv preprint arXiv:2402.11755*, 2024.

[45] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–322, 2024.

[46] Ion Stoica, Matei Zaharia, Joseph Gonzalez, Ken Goldberg, Hao Zhang, Anastasios Angelopoulos, Shishir G Patil, Lingjiao Chen, Wei-Lin Chiang, and Jared Q Davis. Specifications: The missing link to making the development of llm systems an engineering discipline. *arXiv preprint arXiv:2412.05299*, 2024.

[47] Hong Sun, Xue Li, Yinchuan Xu, Youkow Homma, Qi Cao, Min Wu, Jian Jiao, and Denis Charles. Autohint: Automatic prompt optimization with hint generation, 2023.

[48] Zhen Tan, Dawei Li, Song Wang, Alimohammad Beigi, Bohan Jiang, Amrita Bhattacharjee, Mansooreh Karami, Jundong Li, Lu Cheng, and Huan Liu. Large language models for data annotation and synthesis: A survey. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 930–957, 2024.

[49] Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation. *IEEE Transactions on Software Engineering*, 2024.

[50] Masoumeh Taromirad and Per Runeson. A literature survey of assertions in software testing. In *International Conference on Engineering of Computer-Based Systems*, pages 75–96. Springer, 2024.

[51] Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer Verlag, April 2008.

[52] Ke Wang, Jiahui Zhu, Minjie Ren, Zeming Liu, Shiwei Li, Zongye Zhang, Chenkai Zhang, Xiaoyu Wu, Qiqi Zhan, Qingjie Liu, et al. A survey on data synthesis and augmentation for large language models. *arXiv preprint arXiv:2410.12896*, 2024.

[53] Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. Large language models are not fair evaluators. *arXiv preprint arXiv:2305.17926*, 2023.

[54] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool. *arXiv preprint arXiv:2305.04764*, 2023.

[55] Wenda Xu, Guanglei Zhu, Xuandong Zhao, Liangming Pan, Lei Li, and William Wang. Pride and prejudice: LLM amplifies self-bias in self-refinement. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15474–15492, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[56] Zhiyi Xue, Liangguo Li, Senyue Tian, Xiaohong Chen, Pingping Li, Liangyu Chen, Tingting Jiang, and Min Zhang. Llm4fin: Fully automating llm-powered test case generation for fintech software acceptance testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1643–1655, 2024.

[57] Lu Yan, Zhuo Zhang, Guanhong Tao, Kaiyuan Zhang, Xuan Chen, Guangyu Shen, and Xiangyu Zhang. Parafuzz: An interpretability-driven technique for detecting poisoned samples in nlp. *Advances in Neural Information Processing Systems*, 36, 2024.

[58] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1607–1619, 2024.

[59] Dongyu Yao, Jianshu Zhang, Ian G. Harris, and Marcel Carlsson. Fuzzllm: A novel and universal fuzzing framework for proactively discovering jailbreak vulnerabilities in large language models. In *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4485–4489, 2024.

[60] Jiahao Yu, Xingwei Lin, Zheng Yu, and Xinyu Xing. Gptfuzzer: Red teaming large language models with auto-generated jailbreak prompts, 2023.

[61] Jiahao Yu, Yangguang Shao, Hanwen Miao, Junzheng Shi, and Xinyu Xing. Promptfuzz: Harnessing fuzzing techniques for robust testing of prompt injection in llms, 2024.

[62] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726, 2024.

[63] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

[64] Quanjun Zhang, Weifeng Sun, Chunrong Fang, Bowen Yu, Hongyan Li, Meng Yan, Jianyi Zhou, and Zhenyu Chen. Exploring automated assertion generation via large language models. *ACM Transactions on Software Engineering and Methodology*, 2024.

[65] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

A PROMPTS USED IN *PROMPTPEX*

A.1 Output Validation using *PUT*

```
system:
Your task is to very carefully and thoroughly evaluate the given output generated by a
    chatbot to find out if it comply with its description. Since the input is not given
     to you, only check for the rules which can be checked without knowing the input.
    The chatbot description that you must use as the basis for your evaluation are
    provided between the delimiters <DESC> and </DESC>. The description is as follows:
<DESC>
{{ system }}
</DESC>
Here are the guidelines to follow for your evaluation process:
1. **Direct Compliance Only**: Your evaluation should be based solely on direct and
    explicit compliance with the description provided. You should not speculate, infer,
     or make assumptions about the chatbot's output. Your judgment must be grounded
    exclusively in the textual content provided by the chatbot. Do not check for
    anything which requires knowing the input.
2. **Binary Decision on Compliance**: You are required to make a binary decision based
    on your evaluation:
  - Return 'OK' if chatbot output complies with the description (except checks which
      requires knowing the input).
  - Return 'ERR' if there is any non compliance with the chatbot description (except
      checks which requires knowing the input).
3. **Compliance Statement**: Carefully examine the output and determine why the output
    does not comply with the description (except rules which requires knowing the input
    ), think of reasons why the output complies or does not compiles with the chatbot
    description, citing specific elements of the output.
4. **Explanation of Violations**: In the event that a violation is detected, you have
    to provide a detailed explanation. This explanation should describe what specific
    elements of the chatbot's output led you to conclude that a rule was violated and
    what was your thinking process which led you make that conclusion. Be as clear and
    precise as possible, and reference specific parts of the output to substantiate
    your reasoning.
5. **Checking compliance and never correctness**: You are not required to evaluate the
    functional correctness of the chatbot's output as you are not given the input which
     generated those outputs. Your evaluation should focus solely on whether the output
     complies with the chatbot description, if it requires knowing the input, ignore
    that part of the description.
6. **Output guidelines**: For the chatbot's output given to you, first describe your
    thinking and reasoning that went into coming up with the decision then in the next
    line output 'OK' or 'ERR' based on your decision. Output 'OK', if the chatbot's
    output complies with the chatbot description. Output 'ERR', if the chatbot's output
     does not comply with the chatbot description. Do not output anything else.
Example output:
Mention the reason for violation and your thinking went into coming up with it.
ERR
No violation.
OK
By adhering to these guidelines, you ensure a consistent and rigorous evaluation
    process. Be very rational and do not make up information. Your attention to detail
    and careful analysis are crucial for maintaining the integrity and reliability of
    the evaluation.
user:
Chatbot Output: {{ result }}
```

A.2 *IS* Extraction

```
system:
You are an expert in analyzing chatbot functionalities and identifying the requirements
      for their inputs. Given a description of a chatbot's capabilities, your task is to
      specifically extract and list the rules and constraints that will guide the
     creation of valid inputs. Your response should focus solely on input requirements
     and ignore any details related to output generation or other functionalities. Start
      with describing what the input is, is it a question related to programming or is
     it a math problem or something more complex like code or a complete blog post, then
      describe properties of input of this kind and then describe the restrictions for
     the input. Make sure to include all the possible properties of the input and the
     restrictions for the input, for example, the length of the input.
If the chatbot description handles a corner case, for example if the description says
     ignore all the greetings, it means that a greeting is a valid input but the chatbot
      is handling it in a special way which makes it a part of the input domain and
     there must not be a rules against it.
If the input is coming from any kind of file then assume the input will be a string
     containing the content of the file. Only describe the content of the file without
     any details about the file itself.
This input specification will be used for generating tests for the chatbot. Please make
      sure to only think about the input and not the output or how will the chatbot
     respond to the input. If it a possible input, it is a valid input irrespective of
     the output or the chatbot description.
 Please format your response as follows:
 - List each input rule as a clear, independent sentence.
 - Ensure each rule directly relates to the types of inputs the chatbot can accept.
 - Avoid mentioning output details or any assumptions beyond the provided description.
 - Do not add unnecessary details, generate max two rules for each compenent of the
      input.
Focus only on what types of inputs can be given to the chatbot based on its description,
      output each input rule in a line without any bullets, and nothing else.
user:
Chatbot description:
{{context}}
```

A.3 Inverse Rule

```
system:
Given a list of rules provided by the user, generate another list of rules which
     contradicts the given rules semantically.
Generate one inversed rule for each given rule in the given list.
Come up with smart edge case scenarios.
Please ensure that each generated rule is only in a single line.
Output only the generated rules and nothing else.
user:
Rules:
{{rule}}
```

A.4 Extract Rules

```
system:
You are an expert in analyzing chatbot description and extracting rules and constrains
     for output validation. You are given a description for a chatbot. It describes the
     interaction between the user and the chatbot that helps the user achieve their
     goals. Sometimes the description will contain examples. DO NOT provide rules that
     only apply for those examples. Generalize the rules so that they will apply for
     other possible inputs. Ensure the rules are clear, specific and very verbose such
     that they define everything in the rules based on the provided description. Provide
      the rules as meaningful independent sentences that can be easily validated by just
      seeing the output and have all the required information for performing the check.
     Make sure every entity in the rules are provided with a definition and all rules
     must only be about what the output is and should not contain any information about
     how the output should be generated.
{% if num_rules == 0 %}
```

```
Only output all the rules related to the output or response generated by the chatbot
    based on the given description, one in each line and nothing else without any
    bullets or numbering. Do not make any assumptions.
{% else %}
Output at least {{num_rules}} most crucial rules related to the output or response
    generated by the chatbot based on the given description, one in each line and
    nothing else without any bullets or numbering. Do not make any assumptions.
{% endif %}
user:
System prompt: {{input_data}}
```

A.5 Generate Tests

```
system:
You are tasked with developing multiple test cases for an software, given its
    functional and input specification and a list of rules as input. For each rule, you
     must create {{num}} test cases. These test cases must be designed to validate
    whether the software's outputs correctly adhere to a particular rule. These tests
    must be well defined based on the input specifications.
Start with first understanding what is a input for the software using the given input
    specification. Understand what are the different components of a valid input, what
    are the sytax and sematics related constraints. A good test must always be a valid
    input meeting the requirements from the given input specification.
Use the following input specification to understand valid inputs and generate good
    tests: {{input_spec}}
Use the following functional specification of the software to generate the test cases:
    {{context}}
Guidelines for generating test cases:
- Analyze the input specifications to understand the valid input formats, components of
     the input and scenarios for the software.
- If the test case have multiple components, try to use all the components in the test
    case and tag them with their name, like, component name: value
- Develop {{num}} test cases for each rule provided in the list.
- Each test case must be crafted to rigorously assess whether the software's output
    meets the stipulated rule based on the inputs that conform to the provided input
    specification.
- Use valid and realistic input scenarios that fully comply with the input
    specifications and are relevant to the rule being tested.
- Specify clearly in each test case the input given to the software and the expected
    output or behavior that demonstrates adherence to the rule.
- Broadly cover a range of scenarios, including boundary cases, typical cases, and edge
     cases, to thoroughly evaluate the software's adherence to the rule under various
    conditions.
- Never generate similar or redundant test cases
Each test case should adhere to principles of good software testing practices,
    emphasizing coverage, specificity and independence. Critically assess potential
    weaknesses in the software's handling of inputs based on the rule and focus on
    creating diverse test cases that effectively challenge the software's capabilities.
Format your response in a structured CSV format as follows:
- "ruleid": Identifier for the rule being tested.
- "testid": Sequential identifier for each test case under a rule.
- "testinput": Detailed input provided to the software.
- "expectedoutput": Output or behavior expected from the software to affirm rule
    adherence.
- "reasoning": Brief explanation of why this test case is relevant and contributes to
    robust testing of the rule. List the input specification that this test case does
    not follow.
Example CSV layout:
ruleid, testid, testinput, expectedoutput, reasoning
1, 1, "input based on rule 1 scenario 1", "expected outcome demonstrating rule
    adherence", "Explains the relevance and effectiveness of the test and how it
    follows the input specification"
1, 2, "input based on rule 1 scenario 2, examples", "expected response confirming rule",
     "Illustrates how inputs challenge the software and ensure compliance and how is a
    valid test case based on input specification"
```

```
Only output the test cases in the specified CSV format and nothing else. Please make
    sure that the CSV generated is well formed, only have five columns and each value
    in a these columns must only have commas inside quoted value else they will be
    counted as a new column. Do not wrap the output in any additional text or
    formatting like triple backticks or quotes.
Since you will be given {{ num_rules }} rules, you are expected to generated {{
    num_rules * num }} tests, {{ num }} for each given rule.
user:
List of Rules:
{{rule}}
```

B PROMPTS USED IN THE BENCHMARK

B.1 Speech Tag

```
system:
In this task, you will be presented with a sentence and a word contained
in that sentence. You have to determine the part of speech for a given word
and return just the tag for the word's part of speech.
Return only the part of speech tag. If the word cannot be tagged with
the listed tags, return Unknown. If you are unable to tag the word, return
CantAnswer.
Here is the Alphabetical list of part-of-speech tags used in this task: CC:
    Coordinating conjunction, CD: Cardinal number, DT: Determiner, EX: Existential
    there, FW: Foreign word, IN: Preposition or subordinating conjunction, JJ:
    Adjective, JJR: Adjective, comparative, JJS: Adjective, superlative, LS: List item
    marker, MD: Modal, NN: Noun, singular or mass, NNS: Noun, plural, NNP: Proper noun,
     singular, NNPS: Proper noun, plural, PDT: Predeterminer, POS: Possessive ending,
    PRP: Personal pronoun, PRP$: Possessive pronoun, RB: Adverb, RBR: Adverb,
    comparative, RBS: Adverb, superlative, RP: Particle, SYM: Symbol, TO: to, UH:
    Interjection, VB: Verb, base form, VBD: Verb, past tense, VBG: Verb, gerund or
    present participle, VBN: Verb, past participle, VBP: Verb, non-3rd person singular
    present, VBZ: Verb, 3rd person singular present, WDT: Wh-determiner, WP: Wh-pronoun,
     WP$: Possessive wh-pronoun, WRB: Wh-adverb
user:
{{sentenceword}}
```

B.2 Text to P

```
system:
You are a web developer who is formatting a paragraph of text as HTML.
First, please split the paragraph into individual sentences and wrap each sentence with
     a <p> tag.
**Your answer should have at least three <p> tags.**
Then, inside each <p> tag, add one <strong> tag and multiple <em> tags to emphasize key
     words and phrases.
user:
{{text}}
```

B.3 Shakespeare

```
system:
You are a Shakespearean writing assistant who speaks in a Shakespearean style. You help
     people come up with creative ideas and content like stories, poems, and songs that
     use Shakespearean style of writing style, including words like "thou" and "hath".
Here are some example of Shakespeare's style:
- Romeo, Romeo! Wherefore art thou Romeo?
- Love looks not with the eyes, but with the mind; and therefore is winged Cupid
    painted blind.
- Shall I compare thee to a summer's day? Thou art more lovely and more temperate.
Example:
  user: Please write a short text turning down an invitation to dinner.
  assistant: - Dearest, Regretfully, I must decline thy invitation. Prior engagements
      call me hence. Apologies.
```

```
user:
{{question}}
```

B.4 Sentence

```
system:
Rewrite the following sentence to enhance its readability and make it sound more
    conversational. Ensure that the original meaning and factual accuracy are preserved.
     Concentrate on simplifying complex phrases, using language that's easy to relate
    to, and creating a fluid, engaging structure. You're free to change the style,
    wording, and other elements (as specified by the user). Note that this instruction
    is specifically aimed at improving individual sentences, rather than entire
    paragraphs.
For example:
Input: Under the shimmering twilight sky, a curious cat ventured onto the ancient
    cobblestone path, its whiskers twitching with each whisper of the gentle evening
    breeze.
Response: In the enchanting twilight sky, an inquisitive feline embarked on the time-
    honored cobblestone pathway, its whiskers quivering at every murmur of the serene
    evening wind.
Input:
user:
{{text}}
```

B.5 Extract Names

```
system:
Your task is to extract model names from machine learning paper abstracts. Your
    response is an array of the model names in the format [\"model_name\"]. If you don'
    t find model names in the abstract or you are not sure, return [\"NA\"]
user:
Abstract: {{input}}
```

B.6 Elements

```
system:
Extract the important entities mentioned in the text below. First extract all company
    names, then extract all people names, then extract specific topics which fit the
    content and finally extract general overarching themes
Desired format:
Company names: <comma_separated_list_of_company_names>
People names: -||-
Specific topics: -||-
General themes: -||-
user:
Text: {{text}}
```

B.7 Classify

```
system:
A news article can be classified as one of the following categories: World, Sports,
    Business, Sci/Tech.
Examples:
- World: "UN chief urges action on climate change as report warns of 'catastrophe'"
- Sports: "Ronaldo scores twice in Manchester United return"
- Business: "Apple delays plan to scan iPhones for child abuse images"
- Sci/Tech: "SpaceX launches first all-civilian crew into orbit"'
Based on these categories, classify this news article:
user:
{{text}}
```

B.8 Art Prompt

```
system:
Your role is to transform user descriptions into detailed prompts for generating AI
    photos, ensuring each description does not exceed 80 words and is crafted in a
    single paragraph. Focus first on the subjects and their characteristics, then
    detail the timing and lighting, and describe the background. Conclude by conveying
    the feeling the image should evoke. Always generate texts in English, combining
    artistic insight with precise imagery to create impactful AI-generated photos
    within a brief, singular paragraph.
Input from the user:
user:
{{text}}
```

C PROMPTS USED IN EVALUATION

C.1 Baseline

```
system:
You are tasked with developing multiple test cases for an software, use the given
    description to infer its functional and input specification. You must create {{num
    }} distinct and diverse test cases. These test cases must be designed to validate
    whether the software's outputs correctly adhere to description. These tests must be
     well defined as per the description.
Start with first understanding what is a input for the software. Understand what are
    the different components of a valid input, what are the sytax and sematics related
    constraints. A good test must always be a valid input meeting the requirements
    mentioned in the given description.
Use the given description of the software to generate the test cases.
Guidelines for generating test cases:
- Use the description to understand the valid input formats, components of the input
    and scenarios for the software.
- If the test case have multiple components, try to use all the components in the test
    case and tag them with their name, like, component name: value
- Each test case must be crafted to rigorously assess whether the software's output
    meets the stipulated behavior based on the provided software description.
- Use valid and realistic input scenarios that fully comply with the given description.
- Broadly cover a range of scenarios, including boundary cases, typical cases, and edge
     cases, to thoroughly evaluate the software's adherence to the description under
    various conditions.
- Never generate similar or redundant test cases.
- Test cases must never have the output or the expected output, it must only contain
    the input.
Each test case should adhere to principles of good software testing practices,
    emphasizing coverage, specificity and independence. Critically assess potential
    weaknesses in the software's handling of inputs and focus on creating diverse test
    cases that effectively challenge the software's capabilities.
Separate each test case with a new line with "===" as the delimiter. It will help in
    identifying each test case separately. Do not wrap the output in any additional
    text like the index of test case or formatting like triple backticks or quotes.
    Only output the test cases directly separated by "===". Try to generate {{ num }}
    test cases.
user:
Description of the software: {{prompt}}
```

C.2 *IS* Validation

```
system:
Your task is to very carefully and thoroughly evaluate the given input to a chatbot to
    find out if it comply with its input specification, that is, if it is a valid input.
Use the following input specification to evaluate the given input:
<SPEC>
{{input_spec}}
</SPEC>
Here are the guidelines to follow for your evaluation process:
```

```
1. **Direct Compliance Only**: Your evaluation should be based solely on direct and
     explicit compliance with the provided input specification. You should not speculate,
      infer, or make assumptions about the chatbot's description. Your judgment must be
     grounded exclusively in the input specification provided for the chatbot.
2. **Binary Decision on Compliance**: You are required to make a binary decision based
     on your evaluation:
   - Return 'OK' if the given input complies with the input specification.
   - Return 'ERR' if there is any non compliance with the input specification.
3. **Compliance Statement**: Carefully examine the input and determine why the input
     does not comply with the input specification, think of reasons why the input
     complies or does not compiles with the input specification, citing specific rules
     from the input specification.
4. **Explanation of Violations**: In the event that a violation is detected, you have
     to provide a detailed explanation. This explanation should describe what specific
     elements of the input and input specification led you to conclude that a rule was
     violated and what was your thinking process which led you make that conclusion. Be
     as clear and precise as possible, and reference specific parts of the input and
     input specification to substantiate your reasoning.
5. **Output guidelines**: For the input given to you, first describe your thinking and
     reasoning that went into coming up with the decision then in the next line output '
     OK' or 'ERR' based on your decision. Output 'OK', if the input complies with the
     input specification. Output 'ERR', if the input does not comply with the input
     specification. Do not output anything else.
Examples:
Mention the reason for violation and your thinking went into coming up with it.
ERR
No violation.
OK
By adhering to these guidelines, you ensure a consistent and rigorous evaluation
     process. Be very rational and do not make up information. Your attention to detail
     and careful analysis are crucial for maintaining the integrity and reliability of
     the evaluation.
user:
Input: {{test}}
```

C.3 Rule Groundedness

```
system:
You are given a rule and a description of a chatbot.
Your task is to evaluate the rule to determine if it is grounded in the provided
     description.
A rule is considered grounded if it is supported by the information provided in the
     description.
Use the following description to evaluate the rule:
<DESCRIPTION>
{{ description }}
</DESCRIPTION>
Output 'OK' if the rule is grounded in the description. Output 'ERR' if the rule is not
      grounded in the description. Only output the decision as OK or ERR and nothing
     else.
user:
Rule:
{{ rule }}
```

C.4 Task Specification Extraction

```
system:
You are given a description of a chatbot's task. Your task is to extract the intent of
     the chatbot from the given description. The intent is the primary goal or purpose
     of the chatbot. It is the action that the chatbot is designed to perform based on
     the task description.
In the output, provide the extracted intent of the chatbot. Only output the extracted
     intent and nothing else. Do not include any additional information in the output.
user:
```

```
{{ prompt }}
```