Reliability Testing for LLM-Based Systems

As large language models (LLMs) become increasingly integrated into various applications, ensuring their reliability is critical. These systems often take multiple inputs and produce corresponding outputs, each of which must adhere to specific guidelines or criteria. Assessing the reliability of such systems is essential for maintaining trust, safety, and effectiveness. This white paper introduces a framework for conducting reliability tests on LLM-based systems. The framework utilizes validators and verifiers to evaluate the system's behavior across multiple dimensions, providing a comprehensive assessment of its performance.

Section 1: Key Concepts in Reliability Testing

Validators: Ensuring Consistent Behavior

Validators are the foundational elements of the reliability testing framework. They are designed to measure how reliably the system adheres to specific instructions or behaviors. For instance, consider a scenario where an LLM is instructed not to use contractions like "isn't," "doesn't," or "can't." A validator can be implemented to assess how well the model follows this rule.

Simple Validator:

```
Validator(
    name="contraction_validator",
    predicate=lambda o: o.count("'") <= 3,
    minimum_success_percentage=0.95
)
```

Each validator operates on a collection of outputs generated by the system, determining a success percentage that reflects the proportion of outputs meeting the specified criterion. Sometimes validators can have conditional predicates that rely on the input as well:

Conditional Validator

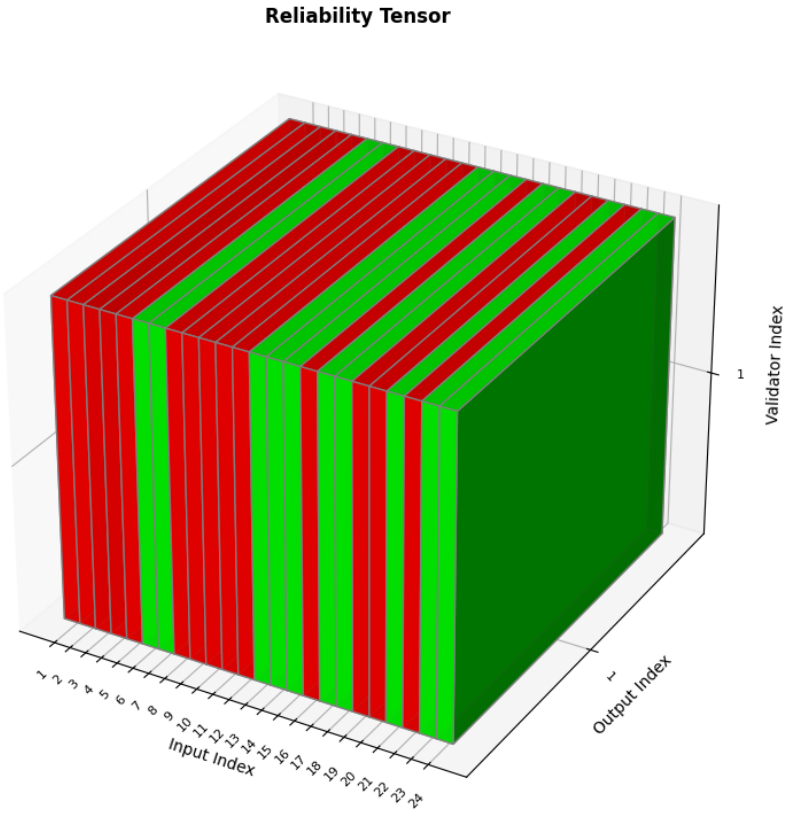
```
Validator(
    name="politeness_validator",
    predicate=lambda i,o: "You're welcome" in o if "Thank you" in i else Tr
    success_percentage=0.90
)
```

Running Binomial Experiments with Validators

Binomial experiments are used to quantify the reliability of the system as determined by a validator. In a Continuous Alignment Testing (CAT) environment, each validator has a minimum success percentage threshold. The outcome of the binomial experiment is compared against this threshold to determine if the system's behavior is reliable. There are two primary methods for conducting these experiments that can be combined into a third:

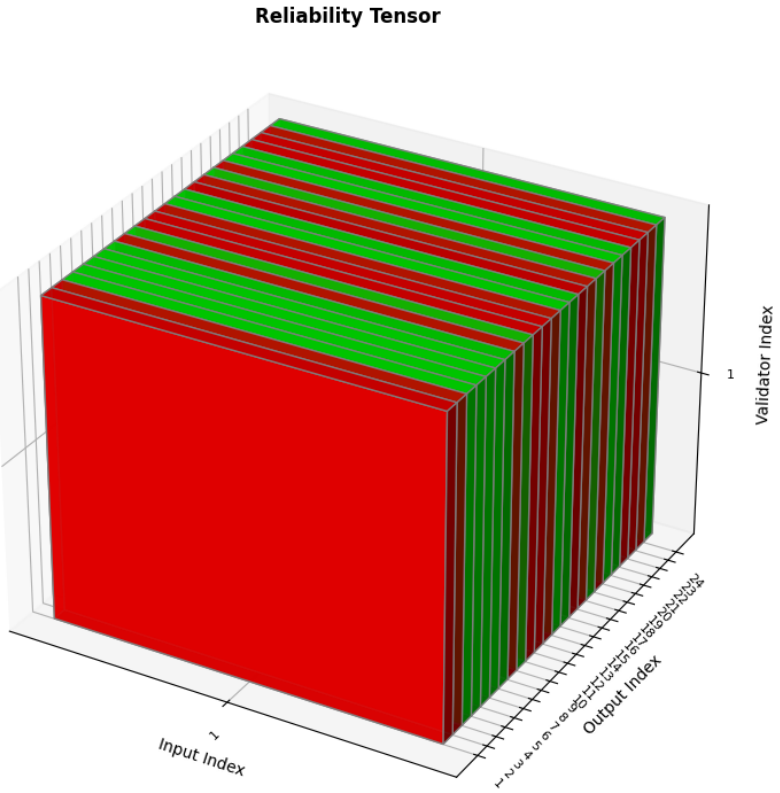
1.1 Varying Inputs, Single Output:

The system generates a single output for each varied input, and the validator assesses the entire collection of outputs, generating a Pass or Fail for each input-output pair (depicted below as green or red markers respectively).



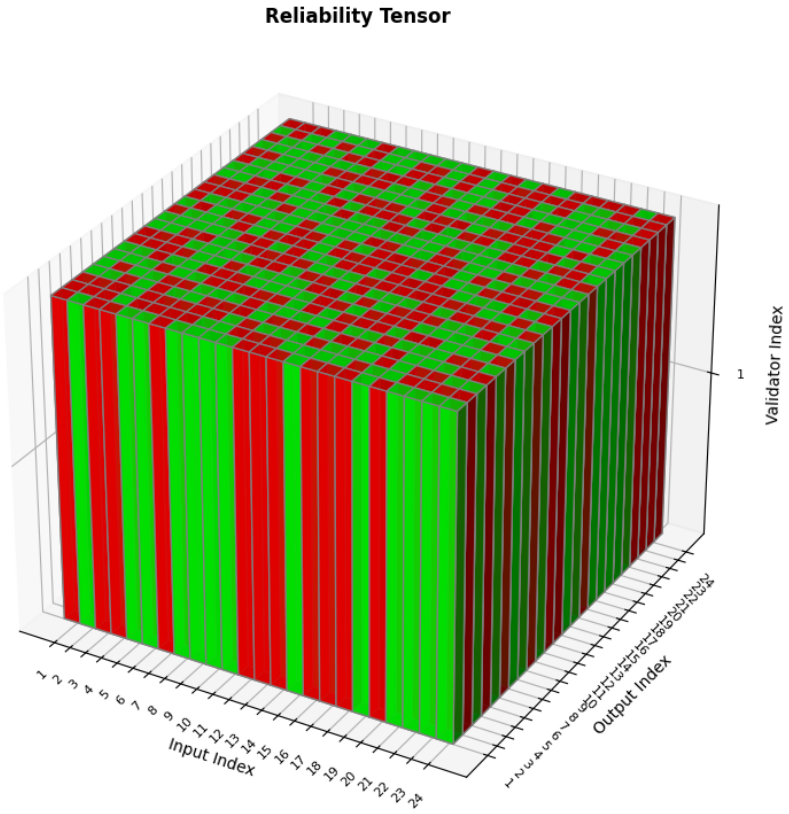
1.2 Fixed Input, Multiple Outputs:

A single input is used to generate multiple outputs, and the validator assesses this set of outputs.



1.3 Varying Inputs, Multiple Outputs

In this approach, the system generates multiple outputs for each varied input, resulting in a comprehensive set of N^2 outputs. This method enables a thorough examination of the system's behavior across a wide range of scenarios, capturing both the variability in inputs and the stochastic nature of output generation.



Aggregating the results

In each of the above cases, and ratio of validator passes to total input-output pairs is compared against the `minimum_success_percentage`, and the Experiment is said to have passed if that percentage is surpassed by the ratio.

Section 2: Scaling Reliability Testing with Multiple Validators

In real-world applications, it is often necessary to assess the reliability of a system across multiple dimensions simultaneously. This requires deploying multiple validators, each designed to measure a specific aspect of the system's behavior. Let's extend this framework to accommodate the use of K validators. This approach allows for a more comprehensive evaluation of the system's reliability, as it accounts for the diverse requirements and constraints

that a system may need to satisfy.

2.1 Understanding the Role of Multiple Validators

When dealing with complex systems, a single validator will not suffice to capture all the nuances of expected behavior. For instance, an LLM may need to meet various criteria such as language style, factual accuracy, ethical considerations, and compliance with specific business rules. Each of these criteria can be represented by a separate validator. The system's overall reliability is then determined by evaluating its performance against all such validators.

Example Use Case:

Consider a content generation system where the LLM must adhere to the following rules:

1. **No Contractions:** Avoid using contractions in the output.
2. **Factual Accuracy:** Ensure that all statements are factually correct.
3. **Ethical Compliance:** Avoid generating content that could be considered biased or offensive.
4. **Tone Consistency:** Maintain a consistent, professional tone throughout the output.

Each of these rules would be represented by a separate validator:

```
Validator(
    name="contraction_validator",
    predicate=lambda o: o.count("'") <= 3,
    minimum_success_percentage=0.95
)

Validator(
    name="factual_accuracy_validator",
    predicate=lambda o: is_factually_correct(o),
    minimum_success_percentage=0.98
)

Validator(
    name="ethical_compliance_validator",
    predicate=lambda o: is_ethical(o),
    minimum_success_percentage=0.99
)

Validator(
    name="tone_consistency_validator",
    predicate=lambda o: is_tone_consistent(o),
    minimum_success_percentage=0.97
)
```

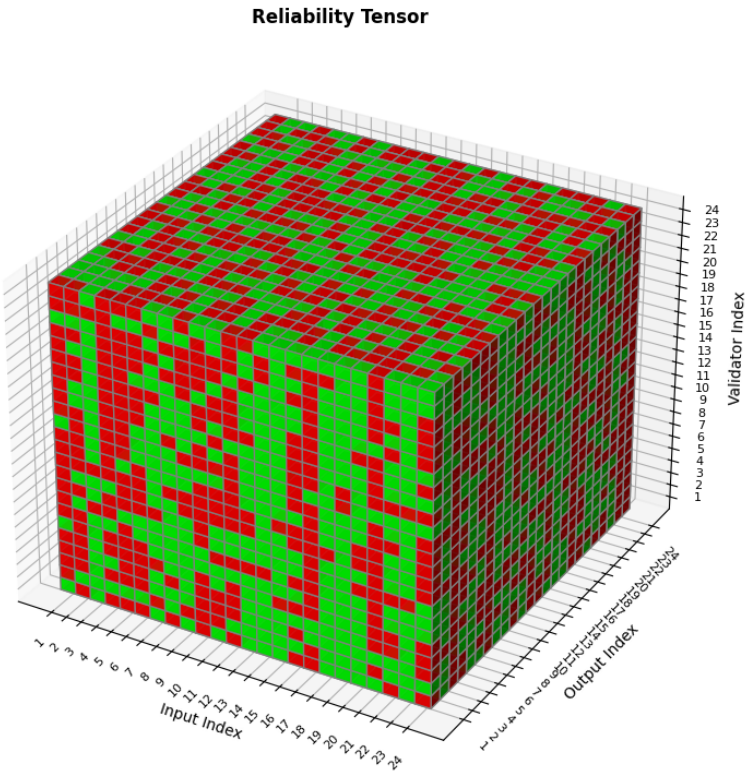
2.2 Generalizing to a Tensor Framework for Reliability Analysis

When extending reliability testing to include multiple inputs, multiple outputs per input, and multiple validators, the system's performance can be represented as a three-dimensional tensor. This **Reliability Tensor** captures the interplay between inputs, outputs, and validators, allowing for a nuanced analysis of the system's reliability.

Constructing the Reliability Tensor

The Reliability Tensor R can be defined with dimensions corresponding to:

- **Input Dimension:** Represents the set of varied inputs $\{input_1, input_2, \ldots, input_N\}$
- **Output Dimension:** Represents the multiple outputs generated per input, $\{output_1, output_2, \ldots, output_M\}$
- **Validator Dimension:** Represents the set of validators $\{validator_1, validator_2, \ldots, validator_K\}$



Each element $R_{i,j,k}$ in the tensor represents the result (pass or fail) of validator k applied to output $output_j$ generated from input $input_i$.

```
R[i][j][k] = Result of validator k on output_j from input_i
```

With this we can define the **Per-Validator Success Rates:** For each validator, aggregate results across inputs and outputs to measure how well the system performs regarding a specific criterion.

```
  Success Percentage for Validator k = Aggregate_{i,j} R[i][j][k]
```

Section 3: Running Experiments

Building upon the concepts introduced in Sections 1 and 2, this section describes how to conduct reliability experiments in practice. The overarching goal is to generate outputs for a set

of inputs, validate each output against one or more validators, and persist the results for future analysis. This process not only yields immediate insights into the system's reliability but also provides a systematic way to track performance over time and across different versions of the system.

3.1 Defining and Configuring an Experiment

An Experiment serves as the central mechanism for orchestrating reliability tests. It takes the following primary components: 1. Experiment Name: A unique identifier for logging and database storage. 2. Validators: A collection of one or more Validator objects, each encapsulating a specific reliability criterion and minimum success percentage. 3. Inputs: The set of prompts or data samples to be fed into the system under test. 4. Outputs (Optional): A pre-computed set of outputs, if they already exist. Otherwise, a generator function can be supplied to create them dynamically. 5. Number of Outputs to Generate: The number of outputs to generate per input in the event that no pre-computed outputs are provided.

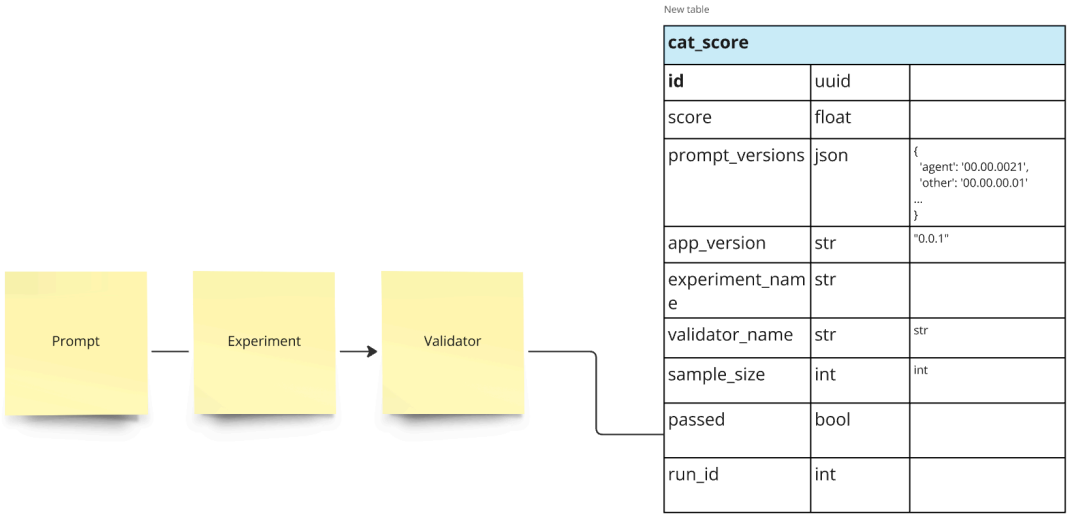
Below is a concise example of creating an experiment using Python code similar to the one in the provided repository:

```
from src.experiment import Experiment, Validator

# Define validators
contraction_validator = Validator(
    name="contraction_validator",
    msp=0.95,
    predicate=lambda inp, out: out.count("'") <= 3
)

politeness_validator = Validator(
    name="politeness_validator",
    msp=0.90,
    predicate=lambda inp, out: ("You're welcome" in out) if "Thank you" in
)

# Define inputs
inputs = [
    "Hello, can you help me?",
    "Thank you for your assistance.",
    "Please do not use contractions in this output."
]

# Define how to generate outputs (a simple placeholder)
def sample_output_generator(input_text):
    # In a real scenario, you'd call your LLM here
    return f"Mock output for: {input_text} (no contractions)."

# Create experiment
exp = Experiment(
    name="my_test_experiment",
    validators=[contraction_validator, politeness_validator],
    inputs=inputs,
    num_to_generate=3,  # generate 3 outputs per input
    output_generator=sample_output_generator
)
```

3.2 Cat Scores

After an Experiment is run it outputs a list of percentages aggregated from its internally maintained Reliability Tensor along the validator index, each such “Cat Score” should be persisted for each Validator you passed to it.

Below is an example of a database table used in a real world scenario



3.3 Persisting Scores for Long-Term Monitoring

Reliability testing is not a one-and-done process; it is best viewed as a **continuous** endeavor. Hence, each time an experiment is run, the resulting CatScore objects can be saved to a database for historical tracking. Over time, the system's performance can be monitored as: - Model architecture or version changes - Prompt engineering strategies evolve - New validators are introduced

3.4 Using Experiments to Drive Prompt Refinement and Coverage

---

Reliability experiments not only serve to monitor system performance but also offer a powerful mechanism for systematically refining prompts to improve overall system reliability. By regularly

running experiments, teams can identify weaknesses in the system's responses and implement targeted improvements to the prompts, enhancing performance across diverse input scenarios. This approach mirrors the Red-Green-Refactor cycle in software testing, where continuous feedback guides iterative enhancements.

Reliability experiments also provide a structured method to "cover" the prompt space of an application. By designing diverse and comprehensive input sets, experiments can test the system's behavior across a wide range of scenarios, exposing edge cases and rare failure modes. This methodical approach ensures that the system is robust against varied user inputs and unexpected situations.

Section 4: Generative Validation

The Time Bottle Neck

While the experiments detailed so far illustrate the power of comprehensive reliability testing, they can also be computationally expensive. In scenarios with both N inputs and M outputs per input, generating the full set of responses to validate may require up to $O(N^2)$ work. This can be impractical in production environments where large numbers of inputs are continuously flowing through the system.

Paying the Time cost in production

However, if your LLM-powered application already maintains a chat history or a record of (input, output) pairs in its own database, you can pre-compute and store the outputs as user interactions occur. By designing your experiments to operate on these collections of inputs and outputs, you can completely eliminate the need to generate responses again specifically for testing. The expensive work of generating responses has already been "paid for" by users at the time they made the requests. Your testing framework then simply queries the database for sufficient sample sizes of each input type and runs the validators against these pre-computed outputs—an operation that typically completes in milliseconds, regardless of how large the collection grows.

This is the core idea behind using the Cat framework like a sidecar to your application: it monitors the database, looks for sample sizes of accumulated input-output data, and once those sample sizes reach a threshold, it runs the experiment, validating all stored outputs in milliseconds, and collapses that newly tested sample into a Cat Score. Because no additional calls to the LLM are needed, your experiments remain near-instantaneous even at scale. This

allows production systems to integrate continuous reliability testing without incurring the steep computational overhead that would otherwise discourage comprehensive validation.

4.1 Generative Conditional Validators

The validators we've discussed so far typically rely on deterministic checks (e.g., string matching, simple Python functions, or straightforward heuristics). While these validators are extremely fast and easy to maintain, they struggle to capture more subjective or context-dependent aspects of language, such as tone, ethics, or factual accuracy.

However, the framework described above suggests that if your application already gathers and stores (input, output) pairs in production, you effectively have "free" test data (since your users have "paid" the computational cost). This opens the door to using a second LLM to label or validate these stored responses, using generative prompts that ask the second LLM to determine compliance. We call these Generative Conditional Validators.

Using separate LLMs for validation has many benefits

1. **Subjective and Context-Heavy Criteria:** Tone, ethics, and accuracy often require deep language understanding and even external knowledge or reasoning. A second LLM can parse and interpret text in ways that a simple Python rule cannot.
2. **Rapid Adaptability:** When corporate or legal policies change, or when new ethical standards must be enforced, updating a second LLM's "validator prompt" is often much faster than writing a new rule-based validator.
3. **Consistency Across Large Datasets:** Human reviewers can be inconsistent or slow for large volumes of user data. A specialized "validator LLM" with a carefully crafted system prompt can provide more consistent judgments at scale.

4.2 Example: Tone and Politeness Check

Suppose you want to ensure your Model Under Test always responds politely, especially if the user's question includes words like "please" or "kindly." Here is how you might implement a tone_validator using a second LLM:

```python
import openai
from src.experiment import Validator

def judge_tone_with_llm(input_text, model_output):
    """
    Sends the (input_text, model_output) to a second LLM
    with a prompt that asks: "Is the model output polite or rude?"
    Returns True if polite, False if not.
    """
    prompt = f"""
    You are an assistant that reviews responses for tone.
    Given the user input and the system's output below:

    USER INPUT:
    {input_text}

    SYSTEM OUTPUT:
    {model_output}

    Evaluate if the SYSTEM OUTPUT is polite. Return only "Yes" or "No".
    """

    response = openai.ChatCompletion.create(
        model="gpt-4",
        messages=[{"role": "system", "content": prompt}],
        temperature=0
    )

    # We expect a single "Yes" or "No"
    answer = response["choices"][0]["message"]["content"].strip()
    return (answer.lower() == "yes")

tone_validator = Validator(
    name="tone_validator",
    predicate=lambda inp, out: judge_tone_with_llm(inp, out),
    minimum_success_percentage=0.95
)
```

Here, the validator calls judgetonewith_llm, which sends a review prompt to another LLM (e.g., GPT-4). That second model decides whether the original output is polite and returns “Yes” or “No”. The validator interprets “Yes” as True (Pass) and “No” as False (Fail).

4.3 Managing Cost and Scalability

Using a second LLM for validation imposes **additional compute costs**, especially if you run these checks in real time. However, recall the strategy from Section 4: - You do **not** need to generate new outputs just for testing. - Leverage the existing (input, output) data collected during normal application usage. - You can run these Generative Conditional Validators offline or in a separate pipeline, perhaps at lower traffic times or on a schedule. - Sample from your user logs to keep costs predictable (e.g., only 10% of data or a randomized subset if volume is high).

4.4 Incorporating Generative Conditional Validators into Experiments

Once you have defined these specialized LLM-based checks, they can be plugged right into your reliability experiments the same way you plug in any other validator. For instance:

```
from src.experiment import Experiment

exp = Experiment(
    name="generative_conditional_experiment",
    validators=[tone_validator, ethics_validator, accuracy_validator],
    inputs=my_database_inputs,  # Provided by your production logs
    num_to_generate=0,   # Because we already have outputs
    precomputed_outputs=my_database_outputs  # Pulled from your logs
)

# Running the experiment
exp.run()
```

Just like other validators, each **Generative Conditional Validator** produces a pass/fail ratio, stored in your aggregated Cat Scores. Over time, you can track how well your system meets more complex criteria, such as ethical compliance or factual accuracy, using a single scoreboard.

4.5 Summary of Benefits

- **Centralized, Automated Review:** A second LLM acts like a “co-pilot” that systematically enforces higher-level or more subjective rules.
- **Immediate Adaptation:** Prompt updates on the validator LLM can quickly reflect new

ethics guidelines or style preferences.

- **Rich Auditing and Monitoring:** Reliability experiments track compliance rates for tone, ethics, or accuracy over time, providing transparent metrics to stakeholders.
- **Minimal Overhead for Production Data:** By leveraging already collected input-output logs, you avoid regenerating expensive outputs solely for validation.

Generative Conditional Validators are a natural extension of the reliability testing framework, enabling you to measure and enforce nuanced constraints on your LLM-based system—well beyond the scope of simple string or pattern checks. This approach elevates reliability testing to address real-world concerns of tone, ethics, and correctness in a systematic, continuous fashion.

Conclusion

Reliability testing is a critical component in the lifecycle of any LLM-based system, ensuring that these increasingly powerful models behave consistently and predictably across diverse scenarios. By treating reliability as a measurable and continuous process, organizations can identify, isolate, and mitigate failures before they affect end-users. The framework introduced in this paper—centered on validators, verifiers, and experiments—offers a structured and extensible approach to evaluating LLM performance. It supports binomial-style experimentation for simple rule checks as well as advanced generative validators that handle more subjective or context-heavy criteria.

Moreover, the concept of a Reliability Tensor enables a holistic view of the system's behavior, capturing results across multiple inputs, outputs, and validators in a single data structure. This approach can be seamlessly extended to real-world production environments by leveraging precomputed user interactions, greatly reducing both time and computational overhead. As LLM-based solutions evolve, this framework remains flexible: new validators, higher-level checks, and domain-specific rules can be integrated with minimal disruption, supporting ongoing innovation while preserving system integrity.

Ultimately, continuous reliability testing aligns LLM systems with organizational goals, ethical guidelines, and user expectations. It allows teams to track long-term performance trends, adapt to changing requirements, and maintain a high level of trust in AI-driven workflows. By adopting a methodical and scalable reliability testing strategy, stakeholders can foster more robust, transparent, and responsible LLM-based applications.