Reliability Testing for LLM-Based Systems

Robert Cunningham

September 2025

Abstract

We present a practical framework for reliability testing of Large Language Model (LLM) based systems. This framework uses validators to measure specific behaviors, employs statistical methods to quantify uncertainty, and tracks reliability across different prompt versions. The approach combines simple binomial statistics with Bayesian updating to provide both point estimates and confidence intervals for system reliability. We demonstrate how prompt version tracking ensures statistical validity when system configurations change.

1 Introduction

As LLMs become integral to production systems, we need robust methods to measure and track their reliability. Traditional software testing doesn't work well for probabilistic systems that can produce different outputs for the same input.

This paper presents a framework that addresses three key challenges:

1. **Measuring Stochastic Behavior:** LLMs are inherently random, so we need statistical approaches
2. **Tracking Configuration Changes:** When prompts change, the system fundamentally changes
3. **Making Deployment Decisions:** We need clear pass/fail criteria with quantified uncertainty

2 Core Concepts

2.1 Validators

Validators are functions that check if an LLM output meets specific criteria. Each validator has:

- A **predicate function** that returns true/false

- A **minimum success percentage (MSP)** threshold
- A **name** for identification

Example Validator:

```
Validator(
    name = "length_validator",
    message = "Output exceeds maximum length",
    predicate = lambda output: len(output.split()) <= 100,
    msp = 0.95 # Must pass 95% of the time
)
```

2.2 Running Tests

For each validator, we:

1. Generate multiple outputs (typically 30-100 samples)
2. Apply the validator to each output
3. Count successes and failures
4. Calculate success rate and confidence intervals

3 Statistical Framework

3.1 Basic Success Rate

The simplest metric is the observed success rate:

$$\hat{p} = \frac{\text{successes}}{\text{total samples}} \tag{1}$$

However, this point estimate doesn't capture our uncertainty, especially with small sample sizes.

3.2 Confidence Intervals

We compute 95% confidence intervals using the normal approximation:

$$\text{CI} = \hat{p} \pm 1.96 \times \sqrt{\frac{\hat{p}(1-\hat{p})}{n}} \tag{2}$$

This tells us the range where the true success rate likely falls.

3.3 Bayesian Approach

For better handling of small samples and edge cases, we use a Bayesian approach with the Beta distribution:

$$\text{Posterior: Beta}(\alpha + \text{successes}, \beta + \text{failures}) \tag{3}$$

Starting with a uniform prior ($\alpha = \beta = 1$), we get:

- **Posterior mean:** $\frac{\alpha+\text{successes}}{\alpha+\beta+\text{total samples}}$
- **95% credible interval:** Using the Beta inverse CDF at 0.025 and 0.975

The key advantage: Bayesian intervals handle edge cases (0% or 100% success) more gracefully than frequentist methods.

4 Prompt Version Tracking

4.1 The Version Problem

When prompts change, the LLM's behavior changes. Statistics collected before the change don't apply to the new configuration. We solve this by tracking prompt versions explicitly.

4.2 Version Hashing

Each unique combination of prompts gets a version identifier:

```
promptVersions = {
    "system_prompt": "v1.2.3",
    "user_template": "v2.0.1",
    "assistant_template": "v1.0.0"
}
versionHash = hash(promptVersions)
```

4.3 Independent Statistics

Each version maintains separate statistics:

- Version A with 95% success rate
- Change prompts → Version B starts fresh
- Version B statistics are computed only from Version B data

This prevents false confidence from mixing statistics across fundamentally different system configurations.

5 Aggregating Results

5.1 Multiple Runs

When we run the same test multiple times, we aggregate results:

```
total_successes = 0
total_failures = 0

for run in test_runs:
    if run.version == current_version:
        successes = run.score * run.sample_size
        failures = run.sample_size - successes
        total_successes += successes
        total_failures += failures

# Compute aggregate statistics
success_rate = total_successes / (total_successes +
    total_failures)
```

5.2 Bayesian Aggregation

With aggregated data, we update our Beta posterior:

$$\text{Beta}(1 + \text{total\_successes}, 1 + \text{total\_failures}) \tag{4}$$

This gives us:

- More precise estimates with more data
- Natural handling of varying sample sizes
- Smooth updates as new data arrives

6 Pass/Fail Decisions

6.1 Conservative Decision Rule

A validator passes if the **lower bound** of its 95% confidence interval exceeds the MSP:

$$\text{PASS} = (\text{Lower CI Bound} > \text{MSP}) \tag{5}$$

This conservative approach ensures we have high confidence before declaring success.

6.2 Example Decision

- Validator MSP: 90%
- Observed success rate: 92%
- 95% CI: [89%, 95%]
- Decision: FAIL (lower bound 89% ¡ 90% MSP)

Even though the point estimate exceeds the threshold, we need more data to be confident.

7 Multiple Validators

7.1 Independent Testing

Each validator is tested independently:

Validator	**Success Rate**	**95% CI**	**MSP**	**Pass?**
length_check	96%	[94%, 98%]	95%	No
no_profanity	99%	[97%, 100%]	99%	No
format_check	93%	[91%, 95%]	90%	Yes

7.2 System-Level Reliability

The probability that an output passes ALL validators:

$$P(\text{all pass}) = \prod_{i=1}^{n} p_i \tag{6}$$

where p_i is the success rate of validator i. This assumes validators are independent.

8 Retry Mechanisms

8.1 Expected Retries

If validators have success rates $p_1, p_2, \ldots, p_n$, the probability of passing all validators is:

$$P(\text{pass all}) = \prod_{i=1}^{n} p_i \tag{7}$$

The expected number of attempts needed:

$$\text{Expected attempts} = \frac{1}{P(\text{pass all})} \tag{8}$$

8.2 Example Calculation

Three validators with success rates: 95%, 90%, 85%

$$P(\text{pass all}) = 0.95 \times 0.90 \times 0.85 = 0.727 \quad (9)$$

$$\text{Expected attempts} = \frac{1}{0.727} \approx 1.38 \quad (10)$$

On average, we need less than 2 attempts for success.

9 Continuous Testing

9.1 Online Updates

As new test results arrive, we continuously update our estimates:

1. Check prompt version
2. If version matches, update statistics
3. If version changed, start fresh statistics
4. Recompute confidence intervals
5. Update pass/fail status

9.2 Version Transitions

When prompts change:

```python
if current_version != previous_version:
    # Start fresh statistics
    successes = 0
    failures = 0
    print(f"Version changed from {previous_version} to {
        current_version}")
    print("Starting new reliability measurements")
```

10 Practical Implementation

10.1 Sample Size Guidelines

For reliable estimates:

- **Minimum:** 30 samples (Central Limit Theorem)
- **Recommended:** 100 samples (tighter confidence intervals)
- **High confidence:** 500+ samples (precise estimates)

10.2 Choosing MSP Thresholds

Common thresholds by criticality:

- **Critical behaviors:** 99% (e.g., no harmful content)
- **Important features:** 95% (e.g., format compliance)
- **Nice-to-have:** 90% (e.g., style preferences)

10.3 Validator Design Tips

1. Make predicates deterministic and fast
2. Use clear pass/fail criteria
3. Test one behavior per validator
4. Name validators descriptively
5. Document expected behavior

11 Visualization

11.1 Reliability Over Time

Track how reliability evolves:

- X-axis: Test run number or timestamp
- Y-axis: Success rate with confidence bands
- Horizontal line: MSP threshold
- Vertical lines: Version changes

This shows:

- Trends in reliability
- Impact of version changes
- Statistical significance of improvements

11.2 Validator Comparison

Compare multiple validators in one view:

- Bar chart of success rates
- Error bars for confidence intervals
- Color coding: green (pass), red (fail), yellow (marginal)

12 Common Patterns

12.1 Conditional Validators

Some validators only apply in certain contexts:

```
Validator(
    name = "citation_validator",
    predicate = lambda input, output:
        "cite sources" not in input.lower() or
        "[" in output and "]" in output,
    msp = 0.95
)
```

12.2 Composite Validators

Combine multiple checks:

```
def composite_check(output):
    length_ok = len(output.split()) <= 100
    format_ok = output.startswith("Answer:")
    tone_ok = "!" not in output # No exclamations
    return length_ok and format_ok and tone_ok

Validator(
    name = "composite_validator",
    predicate = composite_check,
    msp = 0.90
)
```

13 Error Analysis

13.1 Failure Patterns

When validators fail, analyze patterns:

- Group failures by input type
- Identify common failure modes
- Check for systematic biases
- Look for edge cases

13.2 Improving Reliability

Based on failure analysis:

1. Adjust prompts to address failure modes
2. Add examples for edge cases
3. Refine instructions for clarity
4. Consider adjusting MSP if too stringent

14 Extension: Continuous-Valued Validators

14.1 From Binary to Continuous

Traditional validators return binary pass/fail (0 or 1). However, many validation scenarios naturally produce continuous scores between 0 and 1:

```
def partial_compliance_validator(output):
    checks = [
        len(output.split()) <= 100, # Length check
        output.startswith("Answer:"), # Format check
        "please" in output.lower(), # Politeness check
        not contains_jargon(output), # Clarity check
        proper_capitalization(output) # Grammar check
    ]

    # Returns float: 0.0, 0.2, 0.4, 0.6, 0.8, or 1.0
    return sum(checks) / len(checks)

Validator(
    name = "composite_quality",
    predicate = partial_compliance_validator,
    msp = 0.80 # Average score must exceed 80%
)
```

14.2 Mathematical Framework

With continuous validators, each output receives a score $s_i \in [0, 1]$ instead of just 0 or 1.

14.2.1 Mean Score

The average validator score across n samples:

$$\bar{s} = \frac{1}{n} \sum_{i=1}^{n} s_i \tag{11}$$

14.2.2 Confidence Intervals

For continuous scores, we compute confidence intervals using the sample standard deviation:

$$\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(s_i - \bar{s})^2} \tag{12}$$

The 95% confidence interval becomes:

$$\text{CI} = \bar{s} \pm 1.96 \times \frac{\sigma}{\sqrt{n}} \tag{13}$$

14.2.3 Beta Distribution Extension

For Bayesian analysis, we can still use the Beta distribution by treating continuous scores as weighted successes:

$$\text{Effective successes} = \sum_{i=1}^{n} s_i \tag{14}$$

$$\text{Effective failures} = \sum_{i=1}^{n}(1 - s_i) = n - \sum_{i=1}^{n} s_i \tag{15}$$

The posterior becomes:

$$\text{Beta}\left(\alpha_0 + \sum_{i=1}^{n} s_i, \beta_0 + n - \sum_{i=1}^{n} s_i\right) \tag{16}$$

14.3 Practical Examples

14.3.1 Semantic Similarity Validator

```
def semantic_similarity_validator(output, reference):
    # Returns cosine similarity between embeddings (0 to 1)
    output_embedding = get_embedding(output)
    reference_embedding = get_embedding(reference)
    similarity = cosine_similarity(output_embedding,
        reference_embedding)
    return similarity # Float between 0 and 1

Validator(
    name = "semantic_match",
    predicate = semantic_similarity_validator,
    msp = 0.85
)
```

14.3.2 Graded Compliance Validator

```python
def graded_length_validator(output, target=100):
    word_count = len(output.split())

    if word_count <= target:
        return 1.0 # Perfect score
    elif word_count <= target * 1.1:
        return 0.9 # Slightly over
    elif word_count <= target * 1.2:
        return 0.7 # Moderately over
    elif word_count <= target * 1.5:
        return 0.3 # Significantly over
    else:
        return 0.0 # Way over limit
```

14.4 Visualization Improvements

14.4.1 Score Distribution Histogram

Continuous validators enable richer visualizations:

- **X-axis:** Score bins (0-0.1, 0.1-0.2, ..., 0.9-1.0)
- **Y-axis:** Frequency of outputs in each bin
- **Vertical line:** MSP threshold
- **Shading:** Pass zone (green) vs fail zone (red)

This reveals the score distribution shape:

- Bimodal: Outputs cluster at high and low scores
- Normal: Bell curve centered around mean
- Skewed: Most outputs score high (or low)
- Uniform: Scores spread evenly

14.4.2 Heatmap Visualization

For multiple continuous validators:

Output	Length	Format	Tone	Clarity	Avg
Sample 1	0.95	1.00	0.80	0.90	0.91
Sample 2	0.70	0.60	0.95	0.85	0.78
Sample 3	1.00	0.80	0.75	0.70	0.81

Color intensity represents score magnitude, making patterns visible at a glance.

14.5 Advantages of Continuous Validators

1. **Granularity:** Distinguish between "barely passing" (0.51) and "excellent" (0.95)
2. **Smoother Optimization:** Gradients exist for improving scores
3. **Better Aggregation:** Average of 3/5 and 4/5 is meaningful (0.7)
4. **Partial Credit:** Reward outputs that get most things right
5. **Threshold Flexibility:** Can adjust MSP without rewriting validator logic

14.6 Implementation Considerations

14.6.1 Score Calibration

Ensure scores are well-calibrated:

- 0.0 = Complete failure
- 0.5 = Borderline acceptable
- 1.0 = Perfect compliance

14.6.2 Combining Binary and Continuous

Mix validator types in the same system:

```
validators = [
    BinaryValidator("no_profanity", check_profanity, msp=0.99),
    ContinuousValidator("quality", quality_score, msp=0.80),
    BinaryValidator("json_valid", is_valid_json, msp=1.00),
    ContinuousValidator("relevance", relevance_score, msp=0.75)
]
```

14.7 Statistical Power

Continuous validators often require fewer samples for the same statistical power:

- **Binary:** Only captures pass/fail information
- **Continuous:** Captures degree of success/failure

For example, detecting a 5% improvement:

- Binary validator: ∼400 samples needed
- Continuous validator: ∼100 samples needed (depending on variance)

15 Advanced Topics

15.1 Correlated Validators

When validators are correlated (e.g., length and detail level), the independence assumption breaks down. Options:

- Group correlated validators
- Use joint probability models
- Apply correlation corrections

15.2 Adaptive Thresholds

Instead of fixed MSP values, consider:

- Percentile-based thresholds
- Moving averages
- Seasonal adjustments
- User-specific requirements

15.3 A/B Testing

Compare two prompt versions:

1. Run both versions in parallel
2. Collect independent statistics
3. Compare confidence intervals
4. Declare winner when intervals don't overlap

16 Case Study

16.1 Customer Service Bot

A customer service LLM with validators:

Validator	**MSP**	**Purpose**
polite_tone	99%	No rude language
answer_relevant	95%	Addresses question
length_limit	95%	Under 150 words
no_hallucination	98%	Factually correct
format_compliance	90%	Follows template

16.2 Testing Results

After 500 samples:

Validator	**Success Rate**	**95% CI**	**Status**
polite_tone	99.4%	[98.8%, 99.9%]	PASS
answer_relevant	96.2%	[94.9%, 97.5%]	PASS
length_limit	94.8%	[93.2%, 96.4%]	FAIL
no_hallucination	98.6%	[97.7%, 99.5%]	PASS
format_compliance	92.4%	[90.5%, 94.3%]	PASS

The length_limit validator fails because its lower CI (93.2%) is below the 95% MSP.

17 Conclusion

Reliability testing for LLM systems requires:

1. **Statistical thinking:** Embrace uncertainty and use confidence intervals
2. **Version awareness:** Track prompt changes and maintain separate statistics
3. **Conservative decisions:** Use lower confidence bounds for pass/fail
4. **Continuous monitoring:** Update estimates as new data arrives

This framework provides a practical approach to ensuring LLM systems meet reliability requirements while acknowledging their inherent stochastic nature. By combining simple statistics with careful version tracking, teams can deploy LLM systems with confidence.

Future work includes handling validator dependencies, optimizing sample sizes dynamically, and developing more sophisticated aggregation methods for complex multi-validator systems.