Behavioral Contract Testing for AI Features at the Browser Layer

Anonymous Author(s)
Double-blind review submission to IEEE AITest 2026
AI Testing in Practice Track

***Abstract*—AI-powered features—chat assistants, investigation summaries, AI dashboards—are now standard in enterprise web applications, yet the testing ecosystem lacks tools for validating AI behavior at the browser layer where content is rendered to end users. Model evaluation frameworks operate at the API layer, while end-to-end (E2E) test frameworks validate UI mechanics without asserting on AI output semantics. This paper presents *behavioral contract testing*, a novel approach that bridges this gap through declarative YAML contracts enforced by deterministic assertions at the browser layer. We introduce a Playwright-native framework implementing eight custom assertion matchers organized across five validation layers: state machine transitions, timing constraints, structural schema, safety invariants, and semantic similarity. Each layer provides deterministic test oracles for nondeterministic AI output—testing behavioral *boundaries* rather than specific content. We validated the framework against a deployed cybersecurity investigation platform running Amazon Bedrock large language models in production. Across five consecutive validation runs with live, nondeterministic AI responses, the framework achieved a 100% contract pass rate while identifying two previously undetected security issues—PII patterns in AI responses and system prompt fragment leakage—that 174 existing E2E tests had missed. The behavioral contract format serves as a shared specification readable by product managers, executable by test automation, and auditable by compliance teams, enabling compliance-as-code workflows aligned with emerging AI governance requirements.**

***Index Terms*—AI testing, behavioral contracts, test oracles, nondeterministic testing, browser testing, compliance-as-code**

I. Introduction

The rapid integration of large language models (LLMs) into enterprise web applications has created a fundamental testing gap. As of 2026, browser-based AI features—conversational chat agents, AI-generated investigation summaries, intelligent dashboards—are shipping in production across industries including cybersecurity, healthcare, finance, and e-commerce. These features are powered by foundation models such as Amazon Bedrock, OpenAI GPT, and Anthropic Claude, producing nondeterministic natural language output that varies with every invocation.

The testing ecosystem has bifurcated around this challenge. On one side, *model evaluation* tools such as promptfoo [1] and DeepEval [2] assess model quality at the API layer using metrics like hallucination rate, relevance, and factual accuracy. On the other side, *browser automation* frameworks such as Playwright [3], Cypress, and Selenium validate UI mechanics—element visibility, navigation flows, form submissions—without asserting on the semantics of AI-generated content. Neither approach tests what the end user actually experiences: AI-generated content rendered in the browser, subject to the full chain of frontend state management, streaming protocols, error handling, and display logic.

The emerging response to this gap—using an LLM to judge another LLM's output [4]—introduces its own problems. The `@llmassert/playwright` library (v0.6.0, April 2026) provides five LLM-powered matchers, but its design decision that "inconclusive = pass" means security-critical tests silently succeed when the judge model is unavailable or uncertain. Testing nondeterministic systems with another nondeterministic system compounds uncertainty rather than reducing it.

This paper presents *behavioral contract testing*, a fundamentally different approach. Rather than asserting on specific AI output content or delegating judgment to another model, behavioral contracts define deterministic *boundaries* that AI features must satisfy: state machine transitions must occur in declared order, response times must fall within production-derived bounds, structural requirements must be met, and safety invariants must never be violated. The contracts are declarative YAML specifications that serve simultaneously as behavioral documentation, test oracles, and compliance evidence.

We make the following contributions:

1) A **behavioral contract specification** format—declarative YAML files defining five validation layers for AI feature correctness—that provides deterministic test oracles for nondeterministic AI output.
2) A **Playwright-native framework** implementing eight custom `expect()` matchers that enforce behavioral contracts at the browser layer, including a novel DOM state observer capable of sub-frame temporal precision.
3) An **industrial validation** against a deployed cybersecurity investigation platform with live Amazon Bedrock AI, demonstrating 100% contract compliance across five consecutive runs and the detection of two previously unknown security issues.
4) A **multi-audience design** where the same contract artifact is readable by product managers, executable by automation engineers, and auditable by compliance teams—enabling compliance-as-code workflows for emerging AI governance frameworks.

II. Related Work

A. Model Evaluation Frameworks

Tools such as promptfoo [1] and DeepEval [2] evaluate LLM output quality through metrics including hallucination detection, relevance scoring, and factual grounding. These frameworks operate at the model API layer, testing prompt-response pairs in isolation from the browser rendering pipeline. They cannot detect failures that emerge from frontend state management, streaming display logic, or the interaction between AI output and UI components. Our approach is complementary: model evaluation validates what the model *produces*; behavioral contract testing validates what the user *experiences*.

B. LLM-as-Judge Testing

The `@llmassert/playwright` library [4] extends Playwright with five LLM-powered matchers for hallucination detection, PII scanning, tone analysis, format validation, and semantic accuracy—that use GPT as a judge to evaluate AI feature output. While innovative, this approach has three limitations. First, its "inconclusive = pass" design means tests produce false negatives when the judge model is unavailable, rate-limited, or uncertain. Second, the approach introduces nondeterminism into the test oracle itself: the same AI output may be judged differently across runs. Third, LLM judge calls introduce latency and cost that scale linearly with test suite size. Our framework uses exclusively deterministic assertions, ensuring reproducible results independent of external model availability.

C. Conceptual Frameworks

Jones proposed an AI testing pyramid in January 2026 [5], establishing conceptual layers for AI quality assurance. However, no reference implementation accompanied the proposal. The pyramid provides strategic guidance but does not address the practical challenge of constructing deterministic test oracles for nondeterministic browser-layer AI output.

D. Agent Contract Frameworks

The Agent Behavioral Contracts (ABC) framework [6] validates agent orchestration—tool selection, planning sequences, and inter-agent communication—rather than rendered browser output. AgentSpec [7], presented at ICSE 2026, enforces runtime constraints on agent behavior during execution. Both frameworks target the agent orchestration layer, not the browser presentation layer where end users interact with AI features. Our work addresses a different layer of the stack: the point where AI-generated content is rendered in the DOM and displayed to the user.

E. Contract-Based Testing

The concept of software contracts originates with Meyer's Design by Contract [8], where preconditions, postconditions, and invariants define component behavioral specifications. Consumer-driven contract testing tools such as Pact [9] validate API contracts between services. Our behavioral contracts extend this lineage to AI features: the contract specifies behavioral *boundaries* (state transitions, timing bounds, structural requirements, safety invariants) rather than specific input-output pairs, making the approach applicable to nondeterministic systems.

F. State-Based Testing

State machine testing has a long history in software verification [10]. Our contribution applies state machine validation to AI feature lifecycles—where states such as *idle*, *thinking*, *streaming*, and *complete* must transition in declared order—implemented through real-time DOM mutation observation rather than model-level state inspection.

G. Summary

Table I summarizes the capabilities of existing approaches against our framework. No prior approach provides deterministic behavioral contract testing at the browser layer with integrated safety validation and compliance artifact generation.

III. Behavioral Contract Testing

A. The Behavioral Contract

A behavioral contract is a declarative YAML specification that defines what an AI feature must satisfy without constraining what the AI may say. The contract separates *behavioral boundaries*—which are deterministic and testable—from *content*—which is nondeterministic and intentionally unconstrained. Fig. 1 shows a representative contract.

1) Formal Definition: We formalize the behavioral contract as follows.

Definition 1 (Behavioral Contract). A behavioral contract is a tuple $C = (S, T, \Sigma, \Phi, \Psi)$ where:

- $S = (Q, q_0, \delta, F)$ is a finite state machine with states Q (e.g., {idle, thinking, streaming, complete}), initial state q_0, transition function $\delta : Q \times E \to Q$ over triggering events E, and accepting states $F \subseteq Q$.
- $T = (t_{max}, h)$ specifies a maximum response time t_{max} (in milliseconds) with headroom factor $h \in [0, 1]$, such that the effective bound is $t_{max} \cdot (1 + h)$.
- $\Sigma = (l_{min}, l_{max}, R, f)$ specifies structural constraints: minimum and maximum response length, a set of required sections R, and an expected format f.
- $\Phi = \{\phi_1, \ldots, \phi_n\}$ is a set of safety invariants, where each ϕ_i is a predicate over the rendered AI response text. Safety invariants are *negative properties*—they specify what must *never* appear (PII patterns, system prompt fragments, cross-tenant identifiers).
- $\Psi = (sim, \theta)$ specifies a similarity function $sim : \text{Text} \times \text{Text} \to [0, 1]$ and a minimum similarity threshold θ.

Definition 2 (Contract Satisfaction). An AI feature execution trace $\tau = (q_0 \xrightarrow{e_1} q_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} q_n,\ t_r,\ r)$—comprising observed state transitions, response time t_r, and rendered response text r—satisfies contract C (written $\tau \models C$) if and only if:

1) The observed state sequence $(q_0, q_1, \ldots, q_n)$ is a valid path through S.

TABLE I
FEATURE COMPARISON OF AI TESTING APPROACHES

Approach	Browser Layer	Deterministic Oracles	State Machine	Safety Invariants	Compliance Artifacts	No External API Deps.	Contract Spec.
promptfoo [1]	–	✓	–	–	–	–	–
DeepEval [2]	–	✓	–	–	–	–	–
@llmassert/playwright [4]	✓	–	–	–	–	–	–
ABC [6]	–	✓	–	✓	–	✓	✓
AgentSpec [7]	–	✓	–	✓	–	✓	✓
Our framework	✓	✓	✓	✓	✓	✓	✓

```yaml
name: investigation-chat
version: 1.0.0
feature: AI Investigation Chat Agent

stateMachine:
  states: [idle, thinking, streaming, complete]
  transitions:
    - from: idle
      to: thinking
      trigger: user-sends-message
    - from: thinking
      to: streaming
      trigger: ai-begins-response
    - from: streaming
      to: complete
      trigger: ai-finishes-response

timing:
  maxResponseTime: 60000
  headroom: 0.10
  source: p95-production-percentile

structure:
  minResponseLength: 50
  maxResponseLength: 50000
  requiredSections: []
  format: markdown

safety:
  piiDetection:
    enabled: true
    patterns: [ssn, creditCard, apiKey, bearerToken]
    action: fail-and-mask
  systemPromptLeakage:
    enabled: true
    mode: strict
  tenantIsolation:
    enabled: true
    boundaryField: organizationId
  tokenBudget:
    maxTokens: 10000
```

Fig. 1. Behavioral contract for an AI chat feature. Specifies state transitions, timing bounds, structure, and safety invariants—no assertions on specific content.

2) $t_r \leq t_{max} \cdot (1 + h)$.
3) $l_{min} \leq |r| \leq l_{max}$, all sections in R are present in r, and r conforms to format f.
4) $\forall \phi_i \in \Phi : \phi_i(r) = \textit{true}$ (no safety invariant is violated).
5) $sim(r, r_{ref}) \geq \theta$ for a reference description r_{ref}.

Property 1 (Content Independence). For any two execution traces τ_1, τ_2 that share identical state transitions, timing, and structure but differ in AI-generated content, $\tau_1 \models C \Leftrightarrow \tau_2 \models C$ for any contract C where Ψ is omitted. This property is the theoretical foundation of behavioral contract testing: deterministic boundaries remain satisfied regardless of nondeterministic content.

2) Multi-Audience Design: The contract serves four audiences simultaneously:

- A **product manager** reads the state machine and timing requirements as a behavioral specification for the AI feature's expected user experience.
- A **developer** implements against the contract as acceptance criteria, knowing exactly which behavioral boundaries the feature must satisfy.
- A **QA engineer** executes the contract as an automated test suite, with each layer producing deterministic pass/fail results.
- A **compliance officer** maps contract layers to regulatory requirements (e.g., EU AI Act [13], NIST AI RMF [14], SOC 2) as auditable evidence of AI feature governance.

B. Validation Layers

The framework implements five validation layers, each targeting a distinct dimension of AI feature correctness.

1) State Machine Validation: AI features in web applications follow observable lifecycles. A chat agent transitions from *idle* to *thinking* (waiting for model response), to *streaming* (tokens arriving), to *complete* (full response rendered). The `toTransitionThrough` matcher validates that the feature traverses its declared state sequence in order.

State observation is implemented through an `AiStateObserver` injected into the page via Playwright's `addInitScript()` API. The observer attaches a `MutationObserver` to designated DOM elements, recording state transitions with high-resolution timestamps (`performance.now()`). This design handles a critical edge case discovered during validation: AI responses arriving in under 16 milliseconds—faster than a single browser animation frame at 60fps—where the *thinking* indicator and first response token render in the same frame. The observer uses configuration-order scanning within a single `requestAnimationFrame` callback to distinguish states that would otherwise be temporally collapsed.

2) Timing Constraints: The `toMeetTimingContract` matcher enforces response time bounds derived from production performance data. Contracts specify a maximum response time (e.g., 60,000ms based on p95 production latency) with configurable headroom (e.g., 10%). This layer was retroactively validated when a production incident caused 320-second AI processing times exceeding a 300-second frontend Server-

Sent Events (SSE) timeout—exactly the class of regression this assertion is designed to catch before deployment.

3) Structural Schema Validation: The `toMatchAiSchema` matcher validates response structure without asserting on content. Contracts specify minimum and maximum response lengths, required sections, and format constraints (e.g., Markdown). The AI may produce any content within these structural boundaries, preserving the feature's generative flexibility while ensuring responses meet minimum quality thresholds.

4) Safety Invariants: Four matchers enforce security and safety properties:

- `toNotContainPii`: Detects personally identifiable information using deterministic pattern matching with algorithmic validation. Social Security Numbers are validated against known format constraints. Credit card numbers are validated using the Luhn checksum algorithm [11]. API keys and bearer tokens are detected via format-specific patterns. All detected values are *masked* in assertion error output, ensuring test reports are safe to share across teams without exposing sensitive data.
- `toRespectTenantBoundary`: Validates that AI responses do not contain cross-tenant identifiers, preventing data leakage between organizational boundaries.
- `toNotLeakSystemPrompt`: Detects system prompt fragments in AI responses using configurable strict and relaxed matching modes. Strict mode detects exact substring matches; relaxed mode detects semantic patterns associated with prompt evasion techniques.
- `toNotExceedTokenBudget`: Enforces maximum token generation limits to prevent runaway AI responses that could degrade user experience or incur excessive API costs.

5) Semantic Similarity: The `toBeSemanticallySimilar` matcher validates that AI responses maintain semantic alignment with expected behavioral descriptions using TF-IDF cosine similarity [12] combined with Jaccard coefficient overlap. This baseline implementation requires no external API calls, operating entirely within the test process. The matcher exposes a pluggable provider interface for embedding-based similarity (e.g., OpenAI embeddings, sentence-transformers) when higher precision is required.

IV. Architecture and Implementation

A. Framework Architecture

The framework extends Playwright's `expect()` API via the `expect.extend()` mechanism, registering eight custom matchers as first-class assertions. This design requires no modifications to Playwright's core, no custom test runner, and no additional runtime dependencies beyond `js-yaml` for contract parsing. Tests are standard Playwright tests:

```
test('chat satisfies behavioral contract',
  async ({ page, aiContract }) => {
  const contract =
    loadContract('investigation-chat');
  const observer =
    await AiStateObserver.attach(page);

  await page.fill('[data-testid=chat]',
    'Analyze breach data for acme.com');
  await page.click('[data-testid=send]');
  await page.waitForSelector(
    '[data-testid=response-complete]');

  const response = await page
    .textContent('[data-testid=ai-response]');
  const transitions =
    observer.getTransitions();

  // Five validation layers
  expect(transitions)
    .toTransitionThrough(contract.states);
  expect(observer.getTimings())
    .toMeetTimingContract(contract.timing);
  expect(response)
    .toMatchAiSchema(contract.structure);
  expect(response)
    .toNotContainPii(contract.safety);
  expect(response)
    .toNotLeakSystemPrompt(contract.safety);
});
```

Fig. 2. Behavioral contract test execution. Loads a YAML contract and validates each layer against live AI output.

B. State Observation

The `AiStateObserver` is injected into the browser context via `addInitScript()`, executing before any application JavaScript. It attaches a `MutationObserver` to DOM elements matching configurable selectors and maps observed mutations to declared states using a priority-ordered classifier. State transitions are recorded with sub-millisecond timestamps and buffered in a page-accessible variable retrieved by the test via `page.evaluate()`.

The observer implements a critical optimization for high-speed AI responses: when multiple state-indicative DOM mutations arrive within a single animation frame ($<$16.67ms at 60fps), the observer applies configuration-order priority to distinguish states. This prevents temporal collapse—where "thinking" and "streaming" indicators appear simultaneously—from producing false state machine violations.

C. Dual-Mode Execution

The framework supports two execution modes through Playwright's `page.route()` API:

- **Mocked mode** (CI/CD): SSE responses are intercepted and replaced with deterministic fixtures, enabling fast, reproducible contract validation in continuous integration pipelines.
- **Live mode** (nightly/staging): Tests execute against real AI endpoints, validating behavioral contracts against actual nondeterministic model output.

Both modes execute identical contract assertions, ensuring that the same behavioral boundaries are enforced regardless of whether the AI backend is real or simulated.

D. Compliance Artifact Generation

Each test execution generates an AI Bill of Materials (AI-BOM) documenting:

- Which behavioral contracts were validated
- Pass/fail status for each validation layer
- Timing measurements and safety check results
- Regulatory mapping to applicable frameworks (EU AI Act [13], NIST AI RMF [14], SOC 2)

This transforms the test suite from a quality gate into an audit trail, producing compliance evidence as a side effect of normal test execution.

V. Industrial Evaluation

A. Context

The framework was developed and validated at a cybersecurity company operating a cloud-based investigation platform. The platform includes multiple AI-powered features driven by Amazon Bedrock foundation models:

- A **conversational investigation agent** that analyzes breach data, threat indicators, and digital identity information through a chat interface.
- An **AI Insights engine** that generates automated summaries and analytical observations from investigation datasets.
- An **info-stealer analysis module** that processes malware-exfiltrated credential data with AI-assisted triage.

Prior to this work, the platform's E2E test suite comprised 174 Playwright tests covering UI mechanics (navigation, form interaction, data display) but containing zero assertions on AI output behavior, safety, or correctness.

B. Validation Methodology

We designed three behavioral contracts—one per AI feature—and executed each contract five consecutive times against the deployed staging environment with live Amazon Bedrock AI. Each run produced different AI output (non-deterministic generation), testing the core hypothesis that behavioral *boundaries* remain deterministic even when content does not.

C. Results

Table II summarizes the validation results. All 10 contract executions (2 contracts $\times$ 5 runs) passed with a 100% contract compliance rate. A third contract (info-stealer log analysis) was validated during initial development but requires pre-existing data fixtures not available in the automated validation environment; we report only the reproducible automated results. Each run produced distinct AI-generated content—the AI Insights engine produced responses ranging from 4,577 to 4,893 characters across runs—while all validation layers returned deterministic pass results, supporting the hypothesis that behavioral boundaries are content-independent.

TABLE II
Validation Results Across Five Consecutive Runs

Contract	Runs	Pass	Fail	Rate
Investigation Chat	5	5	0	100%
AI Insights	5	5	0	100%
Total	**10**	**10**	**0**	**100%**

TABLE III
Investigation Chat Response Timing (milliseconds)

Metric	R1	R2	R3	R4	R5
AI Response Time	12,007	11,774	13,084	11,013	10,916
Total Test Time (s)	45.5	39.3	36.2	32.2	32.2

1) Security Findings: The safety invariant layer identified two previously undetected security issues that 174 existing E2E tests had not caught:

1) **PII patterns in AI responses.** The `toNotContainPii` matcher detected patterns matching Social Security Number and credit card formats in AI-generated breach analysis output. The breach data domain inherently processes PII, but the AI feature was surfacing raw credential patterns in its natural language summaries rather than masking them. The matcher's error output automatically masked all detected values (e.g., `***-**-4523`), ensuring the test report itself did not propagate sensitive data.
2) **System prompt fragment leakage.** The `toNotLeakSystemPrompt` matcher detected fragments of the system prompt appearing in AI error state responses under specific input conditions. The strict detection mode identified exact substring matches between the configured system prompt and the rendered AI output, revealing an information disclosure vulnerability that could expose internal system architecture to end users.

Neither issue was detectable by the existing test suite, which asserted on element visibility and navigation flow but never inspected AI response content for safety properties.

2) Timing Validation: Tables III and IV show measured variance across runs. Investigation Chat AI response times ranged from 10,916ms to 13,084ms (mean: 11,759ms), all within the 60,000ms contract bound derived from production p95 latency with 10% headroom. AI Insights response lengths ranged from 4,577 to 4,893 characters, confirming that each run produced distinct output while all validation layers passed identically. The timing layer was retroactively validated when a production incident caused 320-second AI processing times exceeding the platform's 300-second SSE timeout—the exact regression class this validation layer is designed to prevent.

D. State Observation Performance

The `AiStateObserver` successfully tracked state transitions across all 10 runs. In the Investigation Chat contract, the *thinking* and *response* states were recorded at identical

TABLE IV
AI INSIGHTS RESPONSE LENGTH ACROSS RUNS (CHARACTERS)

Metric	R1	R2	R3	R4	R5
Response Length	4,577	4,893	4,893	4,665	4,871

timestamps in **every run** (5/5, 100% incidence)—indicating that the AI backend consistently began streaming within the same animation frame as the thinking indicator. For example, Run 1 recorded both states at timestamp 7,298.7ms; Run 5 at 4,836.5ms. The configuration-order priority scanning mechanism correctly resolved all five same-frame collisions without producing false state machine violations. This 100% incidence rate demonstrates that sub-frame state collapse is not an edge case but a routine characteristic of modern LLM response times that any browser-layer AI testing framework must handle.

E. Response Diversity Analysis

To confirm that the AI responses across runs were genuinely nondeterministic (and not cached or templated), we examined two indicators of response diversity.

For Investigation Chat, AI response times varied from 10,916ms to 13,084ms across five runs—a 2,168ms range (19.9% coefficient of variation relative to the mean of 11,759ms). This timing variance reflects different generation paths through the model, as cached or templated responses would produce near-identical latencies.

For AI Insights, response lengths varied from 4,577 to 4,893 characters across five runs—a 316-character range. While the lengths were relatively stable (consistent with the feature generating structured analytical summaries from the same investigation data), all five validation layers returned identical pass results despite the content differences between runs. This supports the content-independence property (Property 1): behavioral boundaries produce deterministic results regardless of the specific content generated.

F. Framework Overhead

The Investigation Chat contract's mean total test execution time was 37.1 seconds across five runs (range: 32.2–45.5s), of which the AI response wait (mean 11,759ms) constitutes the dominant cost. The contract validation itself—state machine checking, timing comparison, structural validation, PII pattern matching with Luhn checksums, and YAML parsing—executes synchronously after the AI response is fully rendered. All contract operations are CPU-bound with no external API calls; the framework's single runtime dependency (`js-yaml`) adds no network latency. The overhead of contract validation is architecturally bounded by local computation on a single response string, which is negligible relative to the seconds-scale AI generation latency that dominates every test run.

G. Regulatory Mapping

With the EU AI Act [13] enforcement beginning August 2026, we mapped each contract validation layer to specific regulatory and governance requirements:

- **State machine validation** maps to EU AI Act Article 14 (human oversight): documented evidence that AI features follow predictable, observable lifecycles.
- **Timing constraints** map to Article 15 (accuracy, robustness, cybersecurity): performance bounds prevent degradation that could affect system reliability.
- **Safety invariants** map to Article 9 (risk management): PII detection, prompt leakage prevention, and tenant isolation are direct risk mitigations.
- **Structural validation** maps to NIST AI RMF [14] Measure 2.6 (AI system performance): response quality thresholds ensure minimum output standards.
- **Compliance artifacts** (AI-BOMs) map to Article 11 (technical documentation) and Article 12 (record-keeping): each test execution produces timestamped, machine-readable evidence of AI feature governance.

This mapping transforms behavioral contracts from a testing tool into a compliance instrument. The same contract that validates AI behavior in CI/CD also generates the documentation required for regulatory audits—without additional tooling or manual documentation effort.

VI. DISCUSSION

A. Lessons Learned

1) Sub-frame timing is the hardest problem: The most technically challenging aspect of behavioral contract testing at the browser layer was not AI nondeterminism—it was temporal precision. Modern LLMs can begin streaming tokens in under 16ms, collapsing multiple observable states into a single render frame. Our initial `MutationObserver` implementation produced false state machine violations until we implemented configuration-order priority scanning.

Our validation data reveals that this is not an edge case: the *thinking* and *response* states were recorded at identical timestamps in 100% of Investigation Chat runs (5/5). The standard browser testing assumption—that observable state changes occur across distinct render frames—is *systematically* violated when AI backends respond at modern inference speeds. Our configuration-order priority approach resolves this by treating DOM mutation ordering within a single frame as a reliable signal, even when temporal separation is not available. We note that this challenge will intensify as model inference latency continues to decrease with hardware acceleration and model optimization.

2) Safety contracts find what UI tests cannot: The two security findings—PII patterns and system prompt leakage—were invisible to 174 existing E2E tests because those tests never inspected AI response *content*. This gap is not unique to our platform. The standard practice in browser E2E testing is to assert on element *presence* and *interaction* ("the chat panel rendered," "the button was clickable") without inspecting

the semantic content of AI-generated text. Safety invariant assertions at the browser layer represent a defense-in-depth strategy: if sensitive data survives model-level guardrails, API-layer filtering, and frontend sanitization, the E2E test is the last checkpoint before the user sees it.

The PII finding is particularly instructive. The cybersecurity platform inherently processes breach data containing real credentials. The AI feature was correctly analyzing this data but surfacing raw patterns (Social Security Numbers, credit card numbers) in its natural language summaries rather than masking them. This is not a model failure—the model was doing what it was asked to do. It is a *presentation layer* failure that only a browser-layer assertion could catch, because the masking responsibility belongs to the rendering pipeline, not the model.

3) Contracts as specification artifacts: An unexpected benefit of behavioral contracts was their utility outside testing. Product managers referenced contracts to understand AI feature behavioral expectations. Developers used contracts as implementation acceptance criteria. This multi-audience utility emerged naturally from the YAML format's readability and the contract's focus on *what* the feature must satisfy rather than *how* it works internally.

This observation suggests a broader insight: the testing community's standard practice of encoding behavioral expectations exclusively in test code creates an information asymmetry where only engineers can read the specification. Declarative contract formats that separate behavioral boundaries from test implementation have the potential to function as *living specifications*—artifacts that remain current because they are executed in CI/CD, unlike documentation that drifts from implementation over time.

4) Why not LLM-as-judge: The framework was designed around deterministic assertions specifically to avoid the failure modes identified in the LLM-as-judge approach. While LLM judges can assess qualities that deterministic matchers cannot (e.g., "is this response helpful?"), our analysis of the competitive landscape—particularly `@llmassert/playwright`'s "inconclusive = pass" design—revealed three categories of risk that are unacceptable for safety-critical assertions. First, *availability risk*: if the judge model is rate-limited, down, or unreachable from CI, every assertion silently passes or is skipped. Second, *reproducibility risk*: the same AI response may receive different judgments across runs, making CI/CD results nondeterministic at the oracle level. Third, *cost risk*: judge calls introduce per-assertion API charges that scale linearly with test suite size, creating an economic disincentive to comprehensive safety coverage. These failure modes motivated the core design decision: safety-critical assertions (PII, prompt leakage, tenant isolation) must be deterministic. The `toBeSemanticallySimilar` matcher provides a controlled exception—using local TF-IDF computation by default, with a pluggable interface for embedding-based providers when teams accept the trade-off of external API dependency for higher semantic precision.

B. Threats to Validity

External validity. The framework was validated against a single production platform with two AI features (a third was validated during initial development but excluded from the automated results due to data fixture dependencies), all powered by Amazon Bedrock. Generalization to other LLM providers (OpenAI, Anthropic, open-source models), other application domains, and other frontend frameworks requires further study.

Internal validity. Five consecutive runs per contract provide preliminary evidence of deterministic boundary behavior but do not constitute statistical significance. Extended validation with larger sample sizes and diverse input distributions would strengthen these findings.

Construct validity. The two security findings were confirmed as genuine issues by the development team, but the absence of false positives across 10 runs does not guarantee a zero false-positive rate at scale. PII detection via pattern matching may produce false positives in domains with different data characteristics than cybersecurity breach analysis.

C. Limitations

The semantic similarity layer currently uses TF-IDF cosine similarity, which captures lexical overlap but not deep semantic equivalence. Embedding-based providers would improve precision but introduce external API dependencies. The framework currently supports text-based AI output; multimodal features (AI-generated images, charts, audio) require additional validation layers not yet implemented.

VII. Conclusion and Future Work

This paper introduced behavioral contract testing—a novel approach to validating AI features at the browser layer through declarative YAML contracts with five deterministic validation layers. The framework addresses a gap in the current testing ecosystem where model evaluation tools test the model and E2E frameworks test the UI, but nothing tests AI behavior as experienced by the end user.

Industrial validation against a deployed cybersecurity platform with live Amazon Bedrock AI demonstrated that behavioral boundaries remain deterministic across nondeterministic AI output: 10 contract executions across two AI features, with a 100% pass rate, two previously undetected security findings, and a 100% sub-frame state collision incidence rate that validated the framework's temporal precision design. The behavioral contract format serves multiple stakeholders simultaneously—product, engineering, QA, and compliance—enabling compliance-as-code workflows relevant to emerging AI governance requirements such as the EU AI Act [13] (enforcement beginning August 2026).

Future work includes: (1) embedding-based semantic similarity for higher-precision semantic validation, (2) visual regression contracts for AI-generated visual content, (3) multi-turn conversation contracts for validating conversational memory and context preservation across dialog turns, (4) formal

verification of contract completeness properties, and (5) empirical evaluation across diverse application domains, LLM providers, and frontend frameworks.

The framework is implemented as a Playwright extension requiring zero modifications to the core framework and a single runtime dependency. Behavioral contract testing demonstrates that the nondeterminism of AI features need not imply nondeterministic testing: by shifting assertions from *content* to *behavior*, we can test what matters—boundaries, safety, and correctness—with the determinism that quality assurance demands.

REFERENCES

[1] I. Webster, "promptfoo: Test your LLM app," GitHub repository, 2024. [Online]. Available: https://github.com/promptfoo/promptfoo

[2] J. Fein, "DeepEval: LLM Evaluation Framework," Confident AI, 2024. [Online]. Available: https://github.com/confident-ai/deepeval

[3] Microsoft, "Playwright: Fast and reliable end-to-end testing for modern web apps," 2024. [Online]. Available: https://playwright.dev

[4] LLMAssert Contributors, "@llmassert/playwright: LLM-powered assertions for Playwright," npm package, v0.6.0, April 2026. [Online]. Available: https://www.npmjs.com/package/@llmassert/playwright

[5] A. Jones, "Testing Pyramid for AI Agents," Block Engineering Blog, January 2026. [Online]. Available: https://engineering.block.xyz/blog/testing-pyramid-for-ai-agents

[6] S. Srinivasan *et al.*, "Agent Behavioral Contracts: Formal Specification and Runtime Enforcement for Reliable Autonomous AI Agents," arXiv preprint arXiv:2602.22302, February 2026.

[7] D. Wang *et al.*, "AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents," in *Proc. ICSE 2026*, arXiv:2503.18666, 2026.

[8] B. Meyer, "Applying 'Design by Contract'," *Computer*, vol. 25, no. 10, pp. 40–51, October 1992.

[9] Pact Foundation, "Pact: Contract testing for microservices," 2024. [Online]. Available: https://pact.io

[10] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," *IEEE Trans. Software Eng.*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[11] H. P. Luhn, "Computer for Verifying Numbers," U.S. Patent 2,950,048, August 1960.

[12] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.

[13] European Parliament, "Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)," *Official Journal of the European Union*, June 2024.

[14] National Institute of Standards and Technology, "AI Risk Management Framework (AI RMF 1.0)," NIST AI 100-1, January 2023.